INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

UNIVERSITÉ D'ORLÉANS

UNIVERSITE D'ORLEANS

ÉCOLE DOCTORALE MATHEMATIQUES, INFORMATIQUE, PHYSIQUE THEORIQUE et INGENIERIE DES SYSTEMES

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLÉANS

THÈSE présentée par :

Damien GROS

soutenue le : **30 JUIN 2014**

pour obtenir le grade de : **Docteur de l'université d'Orléans**

Discipline/ Spécialité : **Informatique**

Protection obligatoire répartie

Usage pour le calcul intensif et les postes de travail

THÈSE DIRIGÉE PAR :

Christian TOINARD	Professeur des universités, INSA Val de Loire

RAPPORTEURS :

Ludovic APVRILLE	Professeur Assistant, Telecom ParisTech
Christian PEREZ	Directeur de recherche INRIA, LIP, ENS Lyon

JURY :

Sébastien LIMET	Professeur des universités, Université d'orléans, Président du jury
Ludovic APVRILLE	Professeur Assistant, Telecom ParisTech
Mathieu BLANC	Docteur, CEA/DAM/Ile de France
Jérémy BRIFFAUT	Maître de conférences, INSA Val de Loire
Christian PEREZ	Directeur de recherche INRIA, LIP, ENS Lyon
Christian TOINARD	Professeur des universités, INSA Val de Loire

Remerciements

Je remercie MM. Ludovic Apvrille et Christian Pérez pour avoir accepté d'évaluer mes travaux en étant rapporteurs de cette thèse. Je remercie M. Sébastien Limet pour avoir accepté de participer à mon jury.

Je tiens à remercier mon directeur de thèse, Christian Toinard, pour son soutien et son aide durant mes recherches, et de m'avoir offert la possibilité de réaliser cette thèse. Je remercie aussi Mathieu Blanc pour m'avoir accueilli au sein de l'équipe SSI du CEA et m'avoir conseillé tout au long de mes 3 années de thèse, et Jérémy Briffaut pour toute l'aide qu'il m'a apporté dans mes travaux.

Je remercie le CEA/DAM/DIF de m'avoir accueilli pendant ces 3 années et de m'avoir fourni les moyens de réaliser toutes les expérimentations nécessaires au cours de mes travaux.

Je tiens à remercier Pierre Clairet, Aurélie Leday et Maxime Fonda de m'avoir accueilli chez eux lors de mes déplacements à Bourges. Je remercie aussi Sandrine pour sa relecture minutieuse de ce mémoire ; Matthieu, Frédéric, Thierry, Pascal et Romain pour les nombreuses discussions enrichissantes et pour toute l'aide qu'ils m'ont fourni durant cette thèse ; les membres de l'équipe SDS du LIFO, permanents et non-permanents pour leur accueil durant mes séjours à Bourges.

Et enfin, je tiens à remercier mes parents, ma famille et mes amis pour leur soutien tout au long de ces 3 années de thèse.

Table des matières

Table des figures **9**

1 Introduction **15**

1.1 Contexte de l'étude 15
1.2 Observateur 16
1.3 Exemples d'attaques visant les systèmes d'exploitation Linux et Windows 17
1.3.1 Exemple d'attaque pour les systèmes Linux 17
1.3.2 Exemple d'attaque pour les systèmes Windows 18
1.3.3 Analyse des deux attaques 19
1.4 Objectif de la thèse 20
1.5 Apports de la thèse 21
1.6 Plan du mémoire 23

2 État de l'art **25**

2.1 Propriétés de sécurité 25
2.1.1 Propriétés générales 25
2.1.1.1 Confidentialité 26
2.1.1.2 Intégrité 26
2.1.1.3 Disponibilité 26
2.1.2 Propriétés dérivées 26
2.1.2.1 Confinement de processus 27
2.1.2.2 Moindre privilège 27
2.1.2.3 Séparation des privilèges 28
2.1.2.4 Non-interférence 28
2.1.3 Discussion 28
2.2 Modèles de contrôle d'accès 29
2.2.1 Le contrôle d'accès discrétionnaire 29
2.2.1.1 Lampson 30
2.2.1.2 HRU 30
2.2.1.3 TAM 31
2.2.1.4 DTAM 31
2.2.2 Le contrôle d'accès obligatoire 32
2.2.2.1 Bell-LaPadula 32
2.2.2.2 Biba 34
2.2.2.3 Role Based Access Control 35
2.2.2.4 Domain and Type Enforcement 36
2.2.3 Discussion 37
2.3 Implantation des mécanismes de contrôle d'accès 38
2.3.1 Linux 38

2.3.1.1 SELinux 38
2.3.1.2 grsecurity 41
2.3.1.3 Tomoyo 42
2.3.1.4 Policy Interaction Graph Analysis 43
2.3.1.5 Politique et règle de contrôle d'accès 45
2.3.2 Windows 45
2.3.2.1 Mandatory Integrity Control 46
2.3.2.2 PRECIP 48
2.3.2.3 Core Force 48
2.3.3 Distributed Security Infrastructure 49
2.3.4 Impact sur les performances 50
2.3.5 Discussion 51
2.4 Conclusion 52

3 Modélisation et répartition d'observateurs 55
3.1 Définition d'un observateur pour les systèmes d'exploitation Linux et Windows . 56
3.1.1 Éléments de base de la protection système 57
3.1.1.1 Modélisation du système observé 61
3.1.1.2 Mode de sécurité 62
3.1.2 Définition d'un observateur : le moniteur de référence 64
3.1.2.1 Phase de pré-décision 65
3.1.2.2 Phase de décision 68
3.1.2.3 Phase de post-décision 68
3.1.3 Répartition des observateurs 69
3.1.3.1 Association 69
3.1.3.2 Localisation 70
3.1.3.3 Redondance 71
3.1.3.4 Discussion 72
3.2 Modélisation d'une politique de contrôle d'accès direct 73
3.2.1 Grammaire de la politique 73
3.2.1.1 La portabilité des noms utilisés pour les contextes 76
3.2.1.2 Modélisation de la problématique des noms 77
3.2.2 Application de la grammaire sur deux modèles de protection pour les systèmes Windows 81
3.2.2.1 Policy-Based Access Control 81
3.2.2.2 Domain and Type Enforcement 84
3.2.2.3 Lien entre la politique d'accès direct et le moniteur de référence 86
3.3 Les techniques de détournement sur les systèmes Windows 87
3.3.1 Explications des techniques 87
3.3.1.1 Détournement de la table des appels système 87
3.3.1.2 *Inline-Hook* 88
3.3.1.3 *filter-driver* 89
3.3.2 Discussion 91
3.3.2.1 Place de l'observateur dans le contrôle d'accès de Windows . . 91
3.3.2.2 Contrôle des *driver* 92
3.4 Discussion 94

4 Répartition d'observateurs en environnement HPC et analyse des performances 97
4.1 Méthode de mesure des performances d'un système réparti d'observateurs 98
4.1.1 Mesures globales 98
4.1.1.1 Mode colocalisé 99
4.1.1.2 Mode distant 100
4.1.2 Comparaison des performances 102
4.1.2.1 Combinatoire 102
4.1.2.2 Un seul observateur 102
4.1.2.3 Deux observateurs 102
4.1.3 Mesures détaillées 103
4.1.3.1 Résultats déduits 105
4.2 Solution pour répartir les observateurs dans un environnement HPC 107
4.2.1 Architecture avec deux observateurs en mode distant 107
4.2.2 Choix des observateurs 108
4.2.2.1 SELinux 108
4.2.2.2 PIGA 108
4.2.2.3 Infiniband 110
4.2.3 Déport du mécanisme de prise de décision 111
4.2.3.1 Le nœud client 111
4.2.3.2 Le proxy serveur 113
4.3 Expérimentations 114
4.3.1 Les moyens de mesure 114
4.3.2 Résultats sur les performances 115
4.3.2.1 Architecture d'expérimentation 115
4.3.2.2 Mesures globales 116
4.3.2.3 Mesures détaillées 118
4.3.2.4 Synthèse des mesures 120
4.4 Amélioration de la sécurité 123
4.4.1 Propriété de sécurité 123
4.4.2 Application de la propriété de sécurité 124
4.5 Discussion 124

5 Contrôle d'accès obligatoire pour Windows et Expérimentations 127
5.1 Politique de contrôle d'accès direct 128
5.1.1 *Path-Based Access Control* 128
5.1.2 Domain and Type Enforcement 130
5.1.3 Modes de fonctionnement 134
5.1.4 Définition des politiques 135
5.1.5 Discussion 137
5.2 Implantation des mécanismes de contrôle d'accès obligatoire 138
5.2.1 Modification de la table des appels système 138
5.2.1.1 Architecture fonctionnelle 139
5.2.1.2 Détournement de la table des appels système 139
5.2.2 Architecture basée sur les *filter-driver* et les *Kernel Callback* 142
5.2.2.1 Architecture globale 142
5.2.2.2 Détournement des flux d'exécution 143
5.3 Expérimentations 144
5.3.1 Violation d'une propriété de confidentialité 145
5.3.1.1 Environnement des expérimentations 145

5.3.1.2 Scénario des attaques 146
5.3.1.3 Violation directe 146
5.3.1.4 Violation indirecte 146
5.3.2 Étude de logiciels malveillants 147
5.3.2.1 Introduction 147
5.3.2.2 Architecture décentralisée 149
5.3.2.3 Protocoles des expérimentations 152
5.3.2.4 Visualisation des résultats et définition des propriétés de sécurité 153
5.4 Discussion 159
5.4.1 Pertinence/complétude de la solution 159
5.4.2 Performances 160

6 Conclusion **161**
6.1 Perspectives 163

7 Bibliographie **165**

I Annexes **169**

A Configuration pour le logiciel `linpack` et utilisation **171**

B Code utilisé pour la réalisation des tests de performances **173**

C Les techniques de détournements **175**
C.1 Détournement de la table des appels système 175
C.2 *Inline Hook* 176
C.3 *Filter-driver* 177

Table des figures

1.1 Schéma présentant la place de l'observateur avec une politique de sécurité vis-à-vis d'une interaction faite par un processus . . . 17
1.2 Schéma résumant les étapes de l'attaque pour l'exploitation d'une faille noyau sous Linux . . . 18
1.3 Schéma résumant les étapes de l'installation d'un *rootkit* nommé *ZeroAccess* . . 19
1.4 Représentation sous la forme de liste des interactions directes . . . 20
1.5 Association d'observateurs . . . 21

2.1 Couverture des propriétés dérivées vis-à-vis des propriétés générales de sécurité . 29
2.2 Représentation de Lampson d'une matrice d'accès . . . 30
2.3 Lois d'application des propriétés de BLP . . . 33
2.4 Lois d'application des propriétés de BLP-restrictives . . . 33
2.5 Modèle **No Read Up/No Write Down** . . . 34
2.6 Lois d'application des propriétés de Biba . . . 34
2.7 Modèle de protection des données : Biba . . . 35
2.8 Schéma d'héritage du modèle $RBAC_1$ . . . 36
2.9 Schéma du modèle DTE . . . 37
2.10 Implantation des modèles RBAC et DTE par SELinux . . . 40
2.11 Fonctionnement de PIGA-CC . . . 45
2.12 Lois d'application des propriétés du MIC . . . 46
2.13 Modèle de protection des données : MIC . . . 47

3.1 Schéma du placement de l'observateur . . . 57
3.2 Écriture d'un fichier malveillant par `Firefox` : Windows et Linux . . . 57
3.3 Écriture détaillée d'un fichier malveillant par un navigateur web . . . 59
3.4 Placement de l'observateur pour détourner le flux d'exécution . . . 59
3.5 Schéma global de détournement des flux pour les deux systèmes . . . 60
3.6 Mode de sécurité : requête/réponse de l'observateur . . . 63
3.7 Mode de sécurité : évidence . . . 63
3.8 Mode de sécurité : notification . . . 63
3.9 Schéma du moniteur de référence . . . 65
3.10 Construction de la trace par le moniteur de référence : ajout de l'horodatage . . . 65
3.11 Construction de la trace par le moniteur de référence : ajout d'information pour l'historique des processus . . . 66
3.12 Construction de la trace par le moniteur de référence : translation en une interaction 66
3.13 Construction de la trace par le moniteur de référence : schéma global . . . 67
3.14 Mode d'association : cascade . . . 69
3.15 Mode d'association : continuation . . . 70
3.16 Mode de localisation : colocalisé . . . 70

3.17 Mode de localisation : distant . . . 70
3.18 Mode de localisation : distant avec plusieurs nœuds et plusieurs observateurs . . 71
3.19 Mode de localisation : distant avec plusieurs nœuds et un seul observateur 71
3.20 Mode de redondance : diffusion des requêtes par le nœud . . . 72
3.21 Mode de redondance : architecture de la forme maître-esclave . . . 72
3.22 Nommage des ressources au format PBAC . . . 75
3.23 Nommage des ressources au format DTE . . . 75
3.24 Transition de rôle modifiant les accès possibles à certaines ressources du système 76
3.25 Correspondance entre une ressource et les emplacements physiques . . . 78
3.26 Correspondance entre une ressource et les emplacements logiques . . . 79
3.27 Représentation de la fonction N_{abs} . . . 79
3.28 Mécanisme de prise de décision du moniteur de référence . . . 87
3.29 Fonctionnement régulier d'un appel système présent dans la table des appels système 88
3.30 Mécanisme de détournement de la table des appels système . . . 88
3.31 Mécanisme d'*inline hook* . . . 89
3.32 Architecture des *filter-driver* en prétraitement . . . 89
3.33 Place de l'observateur en prétraitement . . . 90
3.34 Architecture des *filter-driver* en post-traitement . . . 90
3.35 Empilement de *driver* modifiant la table des appels système . . . 93
3.36 Schéma détaillant la situation de concurrence sur la table des appels système . . . 93

4.1 Prise du temps de référence . . . 98
4.2 Prise du temps avec ajout du premier observateur . . . 99
4.3 Diagramme de séquence illustrant les temps mesurés lors l'ajout d'un seul observateur . . . 99
4.4 Prise du temps global avec deux observateurs en mode colocalisé . . . 100
4.5 Diagramme de séquence illustrant les temps mesurés lors l'ajout de deux observateurs en mode colocalisé . . . 100
4.6 Prise du temps global avec un observateur en mode distant . . . 101
4.7 Diagramme de séquence illustrant le temps global distant lors du déport du second observateur . . . 101
4.8 Prise du temps avec une communication et un observateur en mode distant 103
4.9 Diagramme de séquence illustrant les temps mesurés lors d'une communication complète avec le second observateur en mode distant . . . 104
4.10 Prise du temps pour l'observateur 2 en mode distant . . . 104
4.11 Diagramme de séquence illustrant les temps mesurés lors de la prise de décision du second observateur en mode distant . . . 105
4.12 Architecture décentralisée . . . 107
4.13 Architecture de prise de décision de SELinux . . . 109
4.14 Insertion de PIGA dans l'architecture en mode colocalisé . . . 110
4.15 Schéma d'une communication RDMA sur un lien InfiniBand . . . 111
4.16 Architecture de protection sur un nœud client . . . 112
4.17 Architecture détaillée du contrôle d'accès sur un nœud client . . . 113
4.18 Architecture sur le serveur de sécurité . . . 113
4.19 Diagramme de séquence illustrant la mesure faite au niveau du proxy coté nœud de calcul . . . 119
4.20 Diagramme de séquence illustrant la mesure faite au niveau du proxy serveur pour calculer le temps de prise de décision de PIGA . . . 120

4.21 Diagramme de séquence résumant les différentes mesures effectuées au sein de l'architecture distribuée 121

5.1 Schéma décrivant l'apprentissage d'une politique 136
5.2 Architecture globale 140
5.3 Flux d'exécution classique 141
5.4 Flux d'exécution détourné par la modification de la SSDT 142
5.5 Architecture de détournement basée sur l'utilisation de *filter-driver* et de *kernel callback* 143
5.6 Statistiques sur la politique de contrôle d'accès pour le premier scénario 145
5.7 Schéma du flux indirect conduisant à la violation de la propriété de confidentialité 147
5.8 Instrumentation du système pour l'analyse de logiciel malveillant 150
5.9 Logiciel malveillant écrivant des fichiers dans la corbeille de Windows 154
5.10 `Flash player` écrit dans la Corbeille de Windows 155
5.11 `services.exe` chargeant les fichiers écrits par les logiciels malveillants. 155
5.12 Automate complet des interactions réalisées par le logiciel malveillant 156
5.13 Diagramme d'activité sur l'installation de l'infection 157

Liste des symboles

ACL *Access Control List* : Ensemble des droits d'accès sur une ressource du système, page 30

BLP Bell-LaPadula : Modèle de protection basée sur la confidentialité des données, page 32

DAC *Discretionary Access Control* : Modèle de contrôle d'accès utilisé par la plupart des systèmes d'exploitation laissant la gestion des droits à la discrétion de l'utilisateur, page 29

DTE *Domain and Type Enforcement* : Modèle de protection de haut niveau utilisant les notions de domaine et de type pour abstraire les ressources, page 36

IRP *I/O Request Packets* : Structure en partie opaque du noyau Windows permettant la communication entre les *driver* , page 89

LSM *Linux Security Modules* : Interface au sein des appels système sous Linux autorisant le contrôle des actions, page 39

MAC *Mandatory Access Control* : Contrôle d'accès obligatoire imposant une politique de contrôle d'accès aux utilisateurs du système, page 32

MIC *Mandatory Integrity Control* : Contrôle d'accès obligatoire basé sur les niveaux d'intégrité, page 46

PBAC *Path-Based Access Control* : Modèle de protection basé sur les chemins complets des ressources, page 41

RBAC *Role-Based Access Control* : Modèle de protection reposant sur les rôles des utilisateurs pour donner les autorisations d'accession aux ressources, page 35

SSDT *System Service Dispatch Table* : Table des appels système sur les systèmes Windows, page 140

NTFS *New Technology File System* : Système de fichiers propriétaire utilisé sur les systèmes d'exploitation Windows, page 131

Chapitre 1

Introduction

1.1 Contexte de l'étude

L'actualité montre que les systèmes d'information et plus spécifiquement les systèmes d'exploitation sont la source de vulnérabilités. Le premier système d'exploitation visé est Windows, système le plus utilisé dans le monde pour les ordinateurs personnels et professionnels [1]. Ses principales faiblesses résident dans les applications installées sur le système, qu'elles soient fournies par l'éditeur du système d'exploitation ou par des éditeurs tiers. Ceci est confirmé par les rapports annuels des éditeurs de logiciels antivirus [2]. Nous pouvons ainsi citer des attaques ciblant les logiciels `Adobe Reader` [3] ou `Adobe Flash Player` [4], particulièrement visés par les failles récentes. Ces logiciels sont utilisés par les attaquants pour entrer sur le système d'exploitation afin d'en modifier le comportement, de voler de l'information ou de contribuer à des attaques à grande échelle (botnet, déni de service, etc.).

Les systèmes basés sur Linux font aussi l'objet d'attaques. Ces systèmes sont maintenant largement répandus dans le monde des téléphones mobiles, et de plus en plus présents sur les ordinateurs des professionnels et des particuliers, ce qui en fait une nouvelle cible pour les attaquants. De plus, avec le développement des nouvelles technologies comme l'informatique en nuage (*cloud computing*) ou les grilles de calcul, les menaces se font aussi présentes sur les systèmes distribués. Auparavant, les attaques visaient essentiellement les serveurs pour y récupérer l'information qu'ils hébergeaient. Maintenant, les systèmes répartis offrent des moyens d'attaque supplémentaires et génèrent, par conséquent, des nouvelles problématiques de sécurité.

La protection des systèmes est devenue un enjeu majeur pour les sociétés actuelles, non seulement pour protéger l'intégrité et la confidentialité des données qu'elles possèdent, mais aussi, et c'est de plus en plus le cas, des données ou des services qu'elles hébergent pour le compte de tiers. De plus, la fourniture de ses services est généralement contractuelle, leurs dysfonctionnement entraîne donc des conséquences financières. Les sociétés proposant ce type de service se doivent donc de proposer des moyens de sécurisation efficaces pour protéger les données de leurs clients.

Cette sécurisation du système et des données hébergées passe par la mise en place d'un mécanisme de confiance capable de contrôler les interactions se déroulant au sein même du système. De plus, la confidentialité et l'intégrité des données peuvent être assurées par l'utilisation de moyens cryptographiques forts, que ce soit pour leur stockage ou pour leur transmission sur les réseaux.

1. Statistiques de l'entreprise NetMarketShare : `http://www.netmarketshare.com/operating-system-market-share.aspx`
2. Rapports 2014 de Symantec (`http://www.symantec.com/fr/fr/security_response/publications/threatreport.jsp`) et Sophos (`http://www.sophos.com/en-us/threat-center/security-threat-report.aspx`)
3. CVE-2014-0502 :`http://www.cvedetails.com/cve/CVE-2014-0502/`
4. CVE-2014-0497 : `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0497`

Les systèmes d'exploitation courants sont vulnérables à deux grands vecteurs d'attaque. Le premier vecteur est l'utilisateur. Que ce soit de manière intentionnelle ou non, il peut faire sortir ou modifier de l'information. Par conséquent, il faut contrôler de manière précise ses accès. Dans un contexte de défense, cette vérification passe par la validation du niveau d'habilitation de l'utilisateur, lorsque celui-ci souhaite accéder à une ressource classifiée. La gestion fine des accès aux ressources du système donne l'assurance qu'un utilisateur ne peut accéder qu'aux ressources qui lui sont destinées.

Le second vecteur est l'utilisation de logiciels tiers possédant des failles de sécurité. Les vérifications par l'analyse du code source de l'application ou par le *reverse code engineering* sont en pratique complexes. En plus des logiciels tiers, le système lui-même possède des failles (applicatives ainsi que noyau) qui sont corrigées régulièrement par les développeurs de ces systèmes.

Les systèmes d'exploitation classiques ne sont pas capables de se protéger contre ces deux grands axes d'attaque. Il faut donc proposer d'autres approches. La première méthode concerne les mécanismes cryptographiques capables de chiffrer les données ainsi que d'en vérifier leur intégrité. La seconde méthode concerne les mécanismes de contrôle d'accès, qui vont garantir que les opérations faites par les processus sur le système ne compromettent pas l'intégrité ou la confidentialité. Nous pouvons ainsi citer les antivirus, les antimalwares, les mécanismes de contrôle d'accès obligatoire, etc.

Notre étude va se concentrer sur l'analyse des mécanismes de contrôle d'accès limitant les privilèges des processus, puisque, comme nous le montrerons au cours de cette thèse, c'est ce point qui est la principale source de compromission. En effet, il manque des solutions efficaces, performantes et extensibles capables de limiter et de contrôler les privilèges des processus.

Quel que soit le mécanisme de contrôle d'accès étudié (antivirus, antimalware, etc.), ils possèdent tous une architecture commune composée de deux éléments : le premier élément est un **observateur** et le second est une **politique de sécurité**. Nous nous attacherons à définir ces deux éléments au cours de notre étude.

1.2 Observateur

Le contrôle des accès au sein même du système passe par la mise en place d'un **observateur**. Un observateur est un mécanisme système détournant le flux d'exécution et réalisant des traitements spécifiques vis-à-vis d'une politique de sécurité. Classiquement, ces traitements peuvent être : **auditer**, **interdire**, **autoriser**, **vérifier l'intégrité**, etc. L'observateur se place généralement en espace privilégié (noyau) afin de contrôler la totalité du système et offrir une confiance dans les décisions qu'il prend. Le schéma 1.1 illustre le placement de l'observateur au sein du système.

Un observateur, contrôlant les interactions faites par les processus sur le système, doit commencer par détourner le flux d'exécution régulier des processus afin de l'étudier. Ce détournement doit être sûr car la protection ne doit pas pouvoir être supprimée par un utilisateur. De plus, il est important que ce détournement n'introduise pas un surcoût prohibitif qui dégraderait les performances du système. On peut notamment prendre pour exemple les antivirus qui consomment une grande partie du temps processeur lors de leur analyse.

Dans le cadre d'un observateur contrôlant les accès au sein du système, il doit a minima analyser les **interactions directes**. Une interaction directe est une représentation de l'activité observée sous la forme d'un triplet $(sujet, operation_elementaire, objet)$, qui se représente aussi de la manière suivante :

$$sujet \xrightarrow[operation_elementaire]{} objet$$

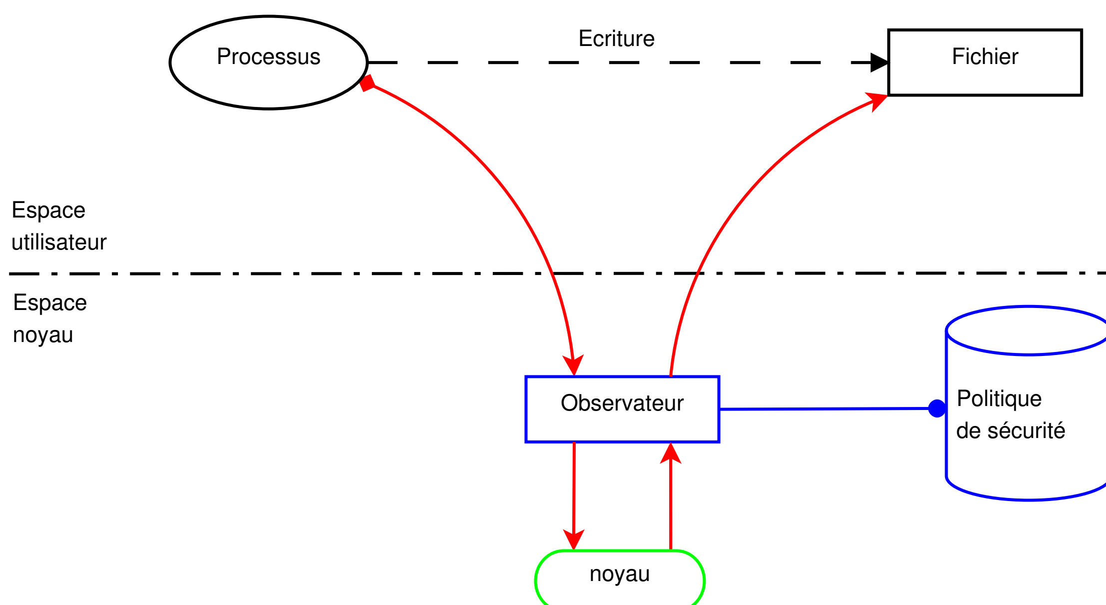

FIGURE 1.1 – Schéma présentant la place de l'observateur avec une politique de sécurité vis-à-vis d'une interaction faite par un processus

Lorsqu'une interaction directe est interceptée, l'observateur recherche le triplet correspondant à une règle dans la politique afin d'exécuter le traitement spécifique associé (autoriser, interdire, auditer, etc.). Sinon il effectue un traitement par défaut, comme lever une alerte. Voyons dans ce qui suit les limites du contrôle des interactions directes.

1.3 Exemples d'attaques visant les systèmes d'exploitation Linux et Windows

À partir de deux exemples concrets pris pour les systèmes d'exploitation Linux et Windows, nous allons illustrer le comportement des attaques pour en déduire les manques dans notre étude.

1.3.1 Exemple d'attaque pour les systèmes Linux

Les systèmes d'exploitation Linux sont aussi exposés aux attaques. On retrouve des attaques exploitant des failles dans des logiciels tiers comme sur les systèmes Windows, mais aussi des attaques réalisées volontairement ou involontairement par les utilisateurs dans le but d'obtenir des privilèges supplémentaires. Dans les deux cas, le but est d'obtenir un accès administrateur (`root`) sur le système pour réaliser des tâches privilégiées.

Il existe des moyens de protection efficaces sur les systèmes Linux. Ces mécanismes permettent de contrôler les interactions directes faites entre les processus et les ressources du système. Cependant, ils ne sont pas capables de contrer les attaques avancées correspondant à l'enchaînement de plusieurs interactions directes, qui prises individuellement peuvent être légitimes, mais qui constituent un **scénario complet** d'attaque.

Nous allons détailler un scénario d'attaque correspondant à l'exploitation d'une faille noyau sur l'appel système `perf_event_open` [5]. Cette vulnérabilité permet d'obtenir un `shell` avec

5. CVE-2013-2094 : `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2094`

un accès `root`. La figure 1.2 illustre un exemple de 9 étapes constituant un ensemble d'interactions directes définissant le scénario complet d'attaque permettant d'obtenir un terminal avec les droits administrateur (un `shell root`).

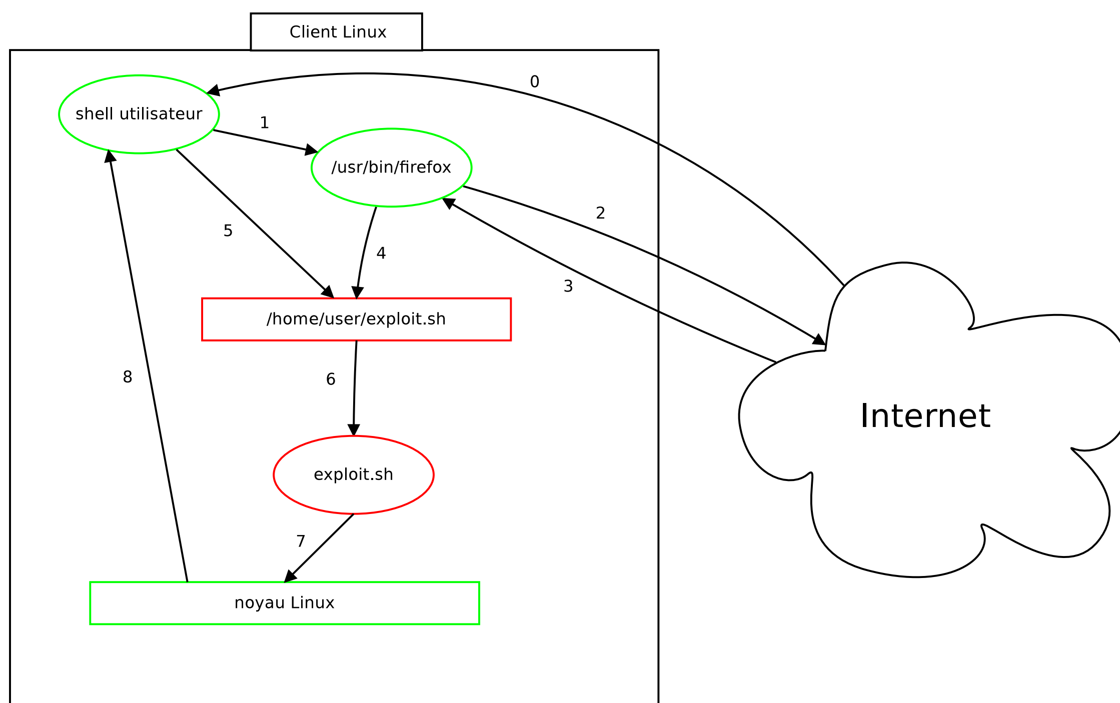

FIGURE 1.2 – Schéma résumant les étapes de l'attaque pour l'exploitation d'une faille noyau sous Linux

Cette attaque se déroule en plusieurs étapes. Nous avons tout d'abord une connexion de la part de l'utilisateur, par exemple via `ssh` (0), l'utilisation d'un navigateur web tel que `Firefox` (1), le téléchargement du logiciel malveillant (2-4) et son exécution (5-8).

Comme l'illustre la figure 1.2, le scénario complet d'attaque comporte 9 étapes composées d'interactions "système" (création de fichier, exécution de programme), mais aussi d'interactions "réseau" (connexion Internet, téléchargement, etc). Ce scénario met donc en jeu plusieurs types d'interactions directes qui prises indépendamment ne constituent pas un risque.

Cette attaque illustre l'exploitation d'une faille du noyau Linux.

1.3.2 Exemple d'attaque pour les systèmes Windows

Les logiciels malveillants (*malware*) ciblant les systèmes d'exploitation Windows sont de plus en plus complexes. Ils utilisent des techniques assez évoluées pour se dissimuler au sein même du système. Il est donc nécessaire de mettre en place des mécanismes de protection avancés contrôlant les processus.

Les attaques actuelles contre les systèmes Windows passent souvent par les navigateurs web et les *plugins* installés. Les *plugins* sont des modules additionnels pour les navigateurs qui apportent des fonctionnalités supplémentaires, comme lire des fichiers *pdf* ou exécuter du code en *flash*. La figure 1.3 montre le cheminement d'une attaque d'un *rootkit* nommé *ZeroAccess* [McAfee, 2013].

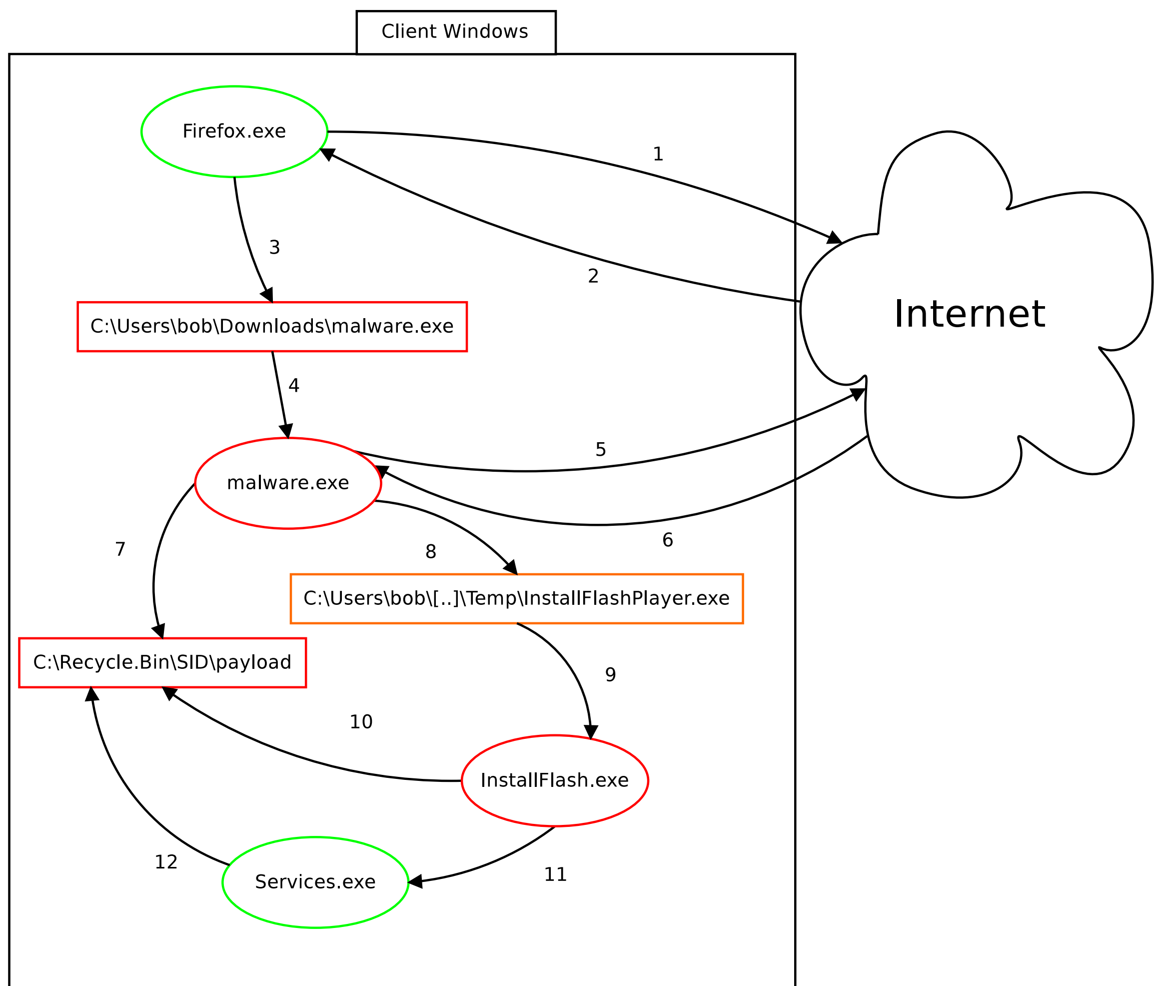

FIGURE 1.3 – Schéma résumant les étapes de l'installation d'un *rootkit* nommé *ZeroAccess*

Afin de compromettre le système, le *malware* va effectuer une suite d'interactions directes. Par l'exploitation d'une faille dans un navigateur ou dans un *plugin* du navigateur, le navigateur va télécharger le *malware* sur le système (1,2). Le *payload*, c'est-à-dire la charge active installant le *rootkit* s'exécute (3) via un installateur modifié de *FlashPlayer* (4-10). Le *rootkit* exploite une faille dans un service système pour finaliser son installation (11, 12). Ce scénario complet est ainsi constitué de 12 interactions directes.

1.3.3 Analyse des deux attaques

Ces deux scénarios complets d'attaque sont proches dans leur construction. Ils sont tous les deux basés sur l'enchaînement de plusieurs interactions directes, c'est-à-dire une succession d'interactions réalisées les unes à la suite des autres de manière logique, menant à la compromission du système.

Pour contrer ce type d'attaque, il est nécessaire d'avoir un observateur capable d'intercepter et de contrôler les interactions directes. Ainsi, l'analyse de ces interactions directes permet de reconstruire le scénario complet sous la forme d'une liste de n interactions directes (figure 1.4).

Par exemple, pour l'attaque concernant le système Windows (figure 1.3), l'interaction numérotée 3 peut se voir sous la forme :

$$Sujet_3 : \textit{Firefox} \xrightarrow[operation_elementaire:open]{} Object_3 : \textit{malware.exe}$$

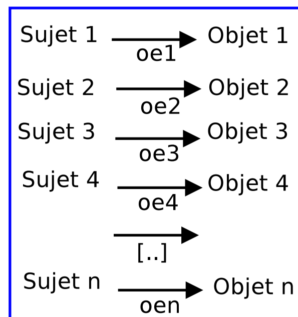

FIGURE 1.4 – Représentation sous la forme de liste des interactions directes

Pour prévenir ce genre d'attaque, on pourrait supprimer (interdire dans la politique de contrôle d'accès) une interaction directe de cette liste. Cependant, certaines interactions sont légitimes comme l'écriture d'un fichier par `Firefox`. On ne peut donc pas interdire une des interactions sans risquer de nuire au fonctionnement du système. Par conséquent, plutôt que d'interdire une des $n - 1$ interactions intermédiaires, il est nécessaire d'analyser le **scénario complet**.

Pour pouvoir contrôler un scénario complet, il faut avoir un observateur réalisant des calculs spécifiques. L'implantation d'un tel observateur, capable à la fois de gérer les interactions directes et de reconstruire temporellement l'activité des processus est difficile à réaliser. En pratique, il est préférable de pouvoir associer différents observateurs. Citons par exemple le défi [ANR, 2009] dans lequel les développeurs ont mis en place deux associations d'observateurs : une première composée de SELinux (mécanisme de contrôle d'accès obligatoire) et iptables (pare-feu) pour contrôler les interactions directes, et une seconde composée de SELinux et de PIGA (voir la section 2.3.1.4) pour le contrôle des scénarios complets.

Afin qu'un observateur, tel que PIGA, puisse analyser le scénario complet, il faut qu'un premier observateur lui communique chaque interaction directe horodatée. Il sera ainsi capable de reconstruire et de contrôler les scénarios complets (figure 1.5).

1.4 Objectif de la thèse

Le travail réalisé au cours de cette thèse vise à améliorer la protection proposée par les systèmes d'exploitation grâce à des approches avancées de contrôle d'accès.

Les mécanismes de protection, qu'ils soient système ou réseau, passent par la mise en place d'un observateur. Cet observateur doit détourner le flux d'exécution et garantir sa conformité vis-à-vis d'une politique de sécurité. Notre étude concerne les observateurs pour les systèmes d'exploitation, leurs répartitions, leurs efficacités et leurs impacts sur les performances du système.

Nous définissons quatre objectifs majeurs pour cette thèse :

1. Définir un modèle d'observateur générique pour les systèmes d'exploitation.
   — Pour mettre en place ce modèle, il faut définir de manière abstraite la notion d'observateur pour les systèmes d'exploitation. Cette notion d'observateur doit être générale et indépendante du système d'exploitation (Linux ou Windows).
   — À partir de cette définition, nous proposerons une implantation fonctionnelle pour les systèmes Windows d'un observateur capable de gérer les interactions directes car c'est sur ce système qu'il y a les manques les plus importants.
2. Nous avons montré au cours de cette introduction qu'il était nécessaire de pouvoir associer et répartir les observateurs pour prévenir efficacement les nouvelles menaces. Il nous faut donc proposer un modèle général de répartition d'observateurs. Cependant, dans des mi-

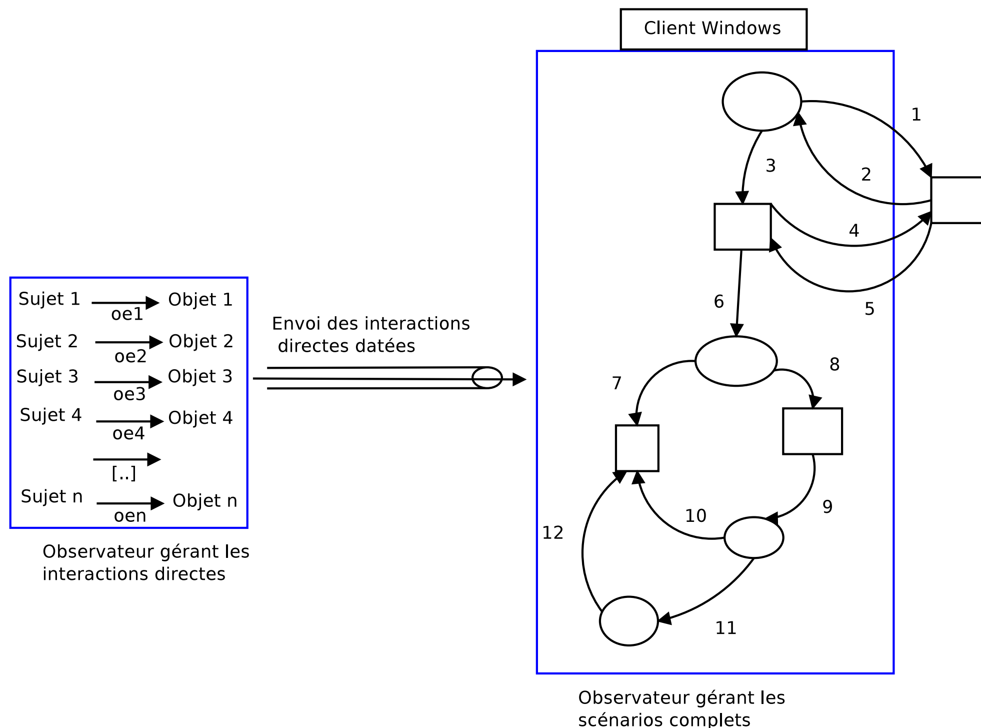

FIGURE 1.5 – Association d'observateurs

lieux où la contrainte en termes de performances est grande, il faut pouvoir évaluer l'impact du modèle de répartition.

3. Il nous faut par conséquent, définir un protocole pour analyser les surcoûts engendrés par l'association d'un ou plusieurs observateurs au sein d'un système et proposer des solutions efficaces pour les réduire.
4. Afin de valider l'efficacité en termes de sécurité, nous développerons deux cas d'usage, l'un pour les systèmes à haute performance sur Linux et l'autre par l'étude des *malware* sur Windows.

1.5 Apports de la thèse

Nous indiquons, au travers de l'état de l'art, qu'il y a un manque de formalisation pour la notion d'observateur. La protection du système passe par la possibilité de pouvoir répartir plusieurs types d'observateurs, réalisant chacun des calculs spécifiques. Dans cette optique, nous proposons une définition générique de la notion d'observateur pour les systèmes d'exploitation. Nous présenterons au travers de notre modélisation, que cette définition ne dépend pas du système d'exploitation sous-jacent. Ainsi, dans le but d'illustrer notre propos, nous montrerons que cette définition s'applique aussi bien aux systèmes Linux qu'aux systèmes Windows.

Afin d'écrire la politique de sécurité d'un observateur, il est nécessaire d'avoir une grammaire. Nous proposons donc une grammaire générique permettant l'écriture des règles de contrôle d'accès directes pour tout type de système. Nous expliquerons comment cette grammaire supporte deux modèles de protection différents s'appliquant à un large ensemble de systèmes différents.

Cette partie répond à la problématique de définition de politique de sécurité générique pour les systèmes.

L'écriture d'une politique de sécurité nécessite d'identifier de manière précise les ressources du système. Nous retrouverons dans la littérature deux méthodes principales de désignation des ressources. La première se base sur le chemin complet de la ressource dans le système de fichiers. Cette méthode, qui a comme principal avantage d'être très usuelle, empêche d'avoir une politique de sécurité parfaitement portable entre les systèmes. La seconde méthode pour désigner les ressources se base sur la définition de types et de domaines. Cette méthode a le principal avantage d'être beaucoup plus portable. Cependant, il est nécessaire d'assurer une correspondance entre les ressources et la définition des types. Nous proposons donc une approche de désignation des ressources applicable à Windows et à Linux et qui rend la politique portable.

Il existe des implantations d'association de plusieurs observateurs : SELinux et PIGA, SELinux et iptables, etc. Néanmoins, il n'existe aucun modèle clair de répartition d'observateurs. Nous définissons donc un modèle exploitable de répartition d'observateurs. Ce modèle permet de définir l'association entre observateurs, mais décrit aussi un modèle pour spécifier la localisation ainsi que le mode de redondance de chaque observateur.

L'intégration et la répartition des observateurs ne doivent pas nuire aux performances du système observé. Mais pour connaître l'impact de l'ajout d'un mécanisme de sécurité sur le système, il faut pouvoir le mesurer. Nous proposons donc une solution pour mesurer l'impact sur les performances de l'ajout et de la répartition d'observateurs. Nous expliquerons comment nous sommes capables d'identifier avec précision les éléments qui nuisent aux performances du système.

Nous décrivons donc un modèle pour mesurer le surcoût engendré par la mise en place d'un ou plusieurs observateurs. Ce modèle est détaillé pour des observateurs colocalisés et distants. Nous proposons l'implantation d'une association de deux observateurs au sein d'un environnement HPC. Afin de réduire les temps de communication entre observateurs, nous déportons le second moniteur en utilisant les technologies spécifiques présentes dans les environnements HPC. Nous illustrons cette partie avec deux observateurs : SELinux pour le contrôle des interactions directes et PIGA pour le contrôle des scénarios complets. Nous montrons ainsi que le déport du second observateur permet de générer moins de latence par rapport à une architecture colocalisée.

Enfin, à partir de la définition générique d'observateur ainsi que de la formalisation de la création d'une politique de sécurité, nous proposons une implantation d'un mécanisme de contrôle d'accès obligatoire pour les systèmes Windows. Nous décrivons l'implantation des deux modèles de politiques de sécurité supportées par notre langage et nous détaillerons les outils que nous avons mis en place pour faciliter l'administration des politiques. Nous détaillerons les mécanismes permettant de détourner de manière efficace les interactions directes sur les systèmes Windows. Nous montrerons que ce mécanisme est capable de contrer les attaques simples, c'est-à-dire en contrôlant les interactions directes, et que, associé avec un second observateur, nous sommes capables de contrôler les scénarios complets. De plus, nous expliquerons qu'il est possible de modifier cet observateur dans le but d'analyser des logiciels malveillants pour en extraire des comportements spécifiques. Cette partie répond à la problématique de manque de mécanisme de contrôle d'accès obligatoire pour les systèmes Windows.

Le tableau 1.1 récapitule les apports de cette thèse.

- couleur verte : les solutions existantes ;
- couleur rouge : les éléments actuellement non satisfaisants que nous développerons dans ce mémoire de thèse ;

	Linux	Windows
Concept d'observateur - Définition commune	Manque une définition générique	
Implantation d'observateurs - Interactions directes	SELinux, grsecurity AppArmor, Tomoyo, SMACK	SEWindows
Répartition des observateurs - Modèle - Implantation HPC	Modèle abstrait de répartition Solution pour les systèmes répartis	
Impact sur les performances - Analyse des performances - Réduction du surcoût	Solutions pour analyser les performances des observateurs Solutions pour réduire le surcoût des observateurs	

TABLE 1.1 – Tableau présentant les apports de la thèse

1.6 Plan du mémoire

Le chapitre 2 fait l'état de l'art des propriétés de sécurité et des principaux modèles de contrôle d'accès. Ensuite, nous traitons des implantations de contrôle d'accès obligatoire existantes sur les systèmes Linux et Windows, en nous focalisant sur les implantations de référence pour chaque système. Cette section se conclut par une synthèse listant les propriétés de sécurité que sont capables de garantir les mécanismes que nous présentons. Enfin, nous nous intéressons à l'impact des mécanismes de contrôle d'accès sur les performances du système. Ce chapitre se conclut en exprimant tout d'abord, les manques des implantations de contrôle d'accès obligatoire existants sur les systèmes Windows et ensuite, en exprimant le manque de formalisme quant à la répartition de plusieurs observateurs au sein des systèmes.

Le chapitre 3 propose une formalisation de la notion d'observateur. Cette formalisation s'appuie sur la modélisation d'un observateur système et sur la définition d'une politique de contrôle d'accès directe. En formalisant la notion d'observateur, nous pourrons spécifier un certain type d'observateur nommé moniteur de référence. De plus, nous nous attacherons à détailler des mécanismes de détournement d'interactions directes pour les systèmes Windows. Nous proposons un modèle de répartitions des observateurs prenant en compte les modes d'association, de localisation et de redondance. La définition d'une politique de contrôle d'accès directe passe par la réalisation d'une grammaire générique spécifiant les traitements de l'observateur lors du détournement d'une interaction. Cependant, la création de règles d'accès passe par une méthode précise pour identifier de manière unique les ressources du système. Pour cela, nous proposerons l'utilisation des noms symboliques absolus indépendants de la localisation. Cette grammaire générique est ensuite appliquée à deux modèles de contrôle d'accès directs, PBAC (*path based access control*) et DTE (*domain and type enforcement*).

Dans le chapitre 4, nous proposons une approche permettant d'analyser le surcoût dû à l'association d'observateur au sein d'un système. Pour cela, nous définissons des points de mesure ainsi qu'une modélisation permettant de connaître le temps pris par chaque élément de l'association. Puis nous implantons une répartition d'observateurs dans un environnement HPC. Nous présenterons une répartition colocalisée et distante pour l'association de SELinux avec PIGA. Nous détaillerons notre architecture permettant de déporter le second observateur, tout en utilisant les technologies spécifiques du HPC pour réduire l'impact d'une telle association sur les performances du système. Enfin, nous appliquerons notre méthode pour mesurer le surcoût de la répartition des observateurs à notre approche HPC. Nous discuterons ainsi la faisabilité d'une répartition d'obser-

vateurs dans les environnements à haute performance sur la base de notre méthode d'évaluation des performances.

Dans le chapitre 5, nous proposons une implantation d'un mécanisme de contrôle d'accès obligatoire pour les systèmes Windows nommé SEWINDOWS basée sur les deux modèles de protection considérés au chapitre 3. Dans une première section, nous définirons deux politiques de contrôle d'accès directes, basées sur les deux modèles de protection que nous avons définis dans le chapitre 3. Nous expliquerons les choix techniques que nous avons dûs faire et notre approche pour implanter une désignation adaptée des ressources. Puis nous détaillerons les deux mécanismes de détournement des flux que nous avons implantés, le premier détournant les appels système en modifiant une table système et le second par l'intégration d'un *filter-driver*. Enfin, nous détaillerons les expérimentations que nous avons faites sur les systèmes Windows. Les premiers tests nous ont permis de valider notre mécanisme de contrôle d'accès en bloquant des attaques. Nous montrons par la suite, qu'en l'associant avec PIGA, nous pouvons gérer des scénarios complets. Puis, nous utilisons notre mécanisme de contrôle d'accès dans le but de pouvoir étudier les logiciels malveillants. Pour cela, nous expliquerons l'architecture et les outils permettant d'exploiter les résultats obtenus.

Enfin le chapitre 6 résume les apports de cette thèse ainsi que les perspectives possibles autour des travaux que nous avons présentés tout au long de ce mémoire.

Chapitre 2

État de l'art

Les problématiques de contrôle d'accès sont présentes sur tous les systèmes d'information. Cette thèse s'intéresse plus spécifiquement aux problématiques de confidentialité et d'intégrité qui font partie intégrante des critères d'évaluation de la sécurité, que ce soit au niveau international [TCSEC, 1985] qu'au niveau européen [ITSEC, 1991]. Ce chapitre présente les travaux de référence en lien avec cette thèse, c'est-à-dire les travaux portant sur les problématiques de contrôle d'accès dans des systèmes hétérogènes.

Ce chapitre s'organise de la manière suivante. Une première partie propose une définition générale des propriétés de sécurité que nous souhaitons garantir au sein d'un système d'information. Cette partie détaille les fondements de la sécurité informatique concernant le contrôle d'accès. La seconde partie s'attache à décrire les différents modèles de contrôle d'accès. Ces modèles ont été définis pour répondre aux problématiques mises en jeu par la définition des propriétés de sécurité. La troisième partie traite de la mise en œuvre de ces modèles de contrôle d'accès. Nous avons séparé cette partie en deux : une première section traite des mécanismes présents sur les systèmes d'exploitation Linux tandis que la seconde s'attache aux systèmes Windows. Enfin, une section traitera de l'impact sur les performances de tels mécanismes. Au sein de la discussion, nous résumerons les faiblesses constatées et nous décrirons le contexte dans lequel nos travaux prennent place.

2.1 Propriétés de sécurité

Les besoins en sécurité s'expriment grâce à des propriétés de sécurité au sein d'une politique. La littérature définit trois grands types de propriétés de référence. Ces propositions sont dépendantes du contexte, des besoins ou encore des utilisateurs. Nous donnons ici une synthèse des définitions exprimées dans les documents de références [TCSEC, 1985, ITSEC, 1991, Bishop, 2003]. Nous avons divisé l'expression des propriétés de sécurité en deux parties : la première partie donne des définitions des trois grands types de propriétés de sécurité, dites générales, alors que la seconde partie exprime les propriétés dérivées de celles-ci.

2.1.1 Propriétés générales

Les trois propriétés de sécurité que nous décrivons dans cette partie sont à la base de la définition des objectifs de sécurité. Ces objectifs de sécurité expriment les besoins des entreprises et des systèmes en termes de sécurité.

2.1.1.1 Confidentialité

La propriété de confidentialité exprime le besoin de prévention des accès non autorisés à l'information. Elle prévient la divulgation de l'information non autorisée comme le définit [Bishop, 2003]. Elle peut être exprimée de la manière suivante :

Définition 2.1.1 ***Confidentialité*** *La confidentialité est la prévention des accès non autorisés à une information.*

La propriété de confidentialité implique que seules les entités autorisées peuvent accéder à l'information. Cette propriété est le plus souvent appliquée dans les milieux de défense assujettis à la classification de l'information. Cette information ne doit être accessible qu'aux personnes disposant de l'habilitation nécessaire.

2.1.1.2 Intégrité

La propriété d'intégrité exprime la prévention de toute modification non autorisée d'une information. Aucun traitement non autorisé ne doit modifier les données que ce soit lors de leur stockage ou lors de leur transmission. Cette propriété implique aussi que les données ne doivent être modifiées ni de manière volontaire ni de manière accidentelle.

Définition 2.1.2 ***Intégrité*** *L'intégrité est la prévention des modifications non autorisées d'une information.*

Tout comme la propriété de confidentialité, la propriété d'intégrité implique que seules les entités autorisées du système ont le droit de modifier l'information.

2.1.1.3 Disponibilité

La propriété de disponibilité repose sur la possibilité d'accès à une donnée ou à un service à tout instant. Cette notion est à rapprocher de la fiabilité d'un système puisque si un service n'est pas accessible, il est considéré comme défaillant. En matière de sécurité informatique, cette défaillance peut se traduire par l'épuisement d'une ressource partagée entre plusieurs entités du système.

Définition 2.1.3 ***Disponibilité*** *La disponibilité est la prévention de la rétention d'information ou d'un service.*

Le contexte d'application peut faire varier le sens de cette définition. En effet, les systèmes critiques ont des besoins très forts en termes de disponibilité alors qu'un site internet aura des besoins moindres. Cette propriété de sécurité se rapprochant plus de la sûreté de fonctionnement que de la sécurité, elle sort du cadre de notre étude.

2.1.2 Propriétés dérivées

Les propriétés générales de sécurité sont extrêmement difficiles à mettre en place. En effet, en pratique, un système est parfaitement intègre et respecte la propriété de confidentialité s'il ne rend aucun service. C'est pour cette raison qu'à partir des propriétés de confidentialité, d'intégrité et de disponibilité, il est possible de dériver de nouvelles propriétés de sécurité plus spécifiques qui sont applicables sur les systèmes actuels. Ces propriétés dérivées sont des cas particuliers des 3 types précédents.

2.1.2.1 Confinement de processus

La problématique du confinement des processus a été écrite par Lampson [Lampson, 1973].

Définition 2.1.4 ***Confinement de processus*** *Le problème du confinement concerne la prévention de la divulgation, par un service (ou un processus), d'informations considérées comme confidentielles par les utilisateurs de ce service.*

Dans la résolution de cette problématique, Lampson énonce les caractéristiques nécessaires à un processus pour qu'il ne puisse pas divulguer d'information. Une des caractéristiques principales est qu'un processus observable ne doit pas stocker d'informations. Si un processus stocke de l'information jugée confidentielle et qu'un utilisateur peut observer ce processus, alors il existe un risque d'échange d'information entre le processus observé et l'observateur.

Cette propriété est impérativement transitive par nature. Si un processus lance un nouveau processus, et qu'il n'est pas possible d'avoir confiance en ce dernier, alors il est nécessaire que ce nouveau processus soit confiné lui aussi.

Le modèle de Lampson décrit dans la pratique la propriété d'*isolation totale*, c'est-à-dire qu'un processus ne peut pas échanger d'information par quelque moyen que ce soit avec d'autres processus. Cette propriété est pratiquement inapplicable dans les systèmes d'exploitation classiques, puisque les processus partagent des ressources comme le processeur, le matériel, etc. Il existe, de plus, des moyens de communication entre processus appelés les canaux cachés.

Définition 2.1.5 ***Canaux cachés*** *Les canaux cachés sont des moyens de communication entre deux entités actives du système. Ce moyen de communication n'est pas un moyen légitime de communication dans la conception du système.*

Prenons comme un exemple deux processus p_1 et p_2 qui ne peuvent pas communiquer entre eux. Le processus p_1 souhaite envoyer de l'information au processus p_2, pour cela, il va trouver une ressource que les deux peuvent partager, comme un élément du système de fichiers. Il va ainsi pouvoir jouer soit sur le nom du fichier pour transmettre une information, soit sur le contenu du fichier. Par exemple, en créant des fichiers nommés 0, 1 et end, il va pouvoir transmettre, sous la forme d'un code défini entre les deux processus, un message chiffré à destination du second processus.

2.1.2.2 Moindre privilège

Le principe de moindre privilège a été défini par [Saltzer et Schroeder, 1975]. Il établit qu'une entité active du système ne devrait s'exécuter sur le système qu'avec les privilèges nécessaires à la réalisation de sa tâche. Ce principe s'applique aussi bien aux processus qu'aux utilisateurs du système.

Définition 2.1.6 ***Moindre privilège*** *Le principe de moindre privilège est la réduction des droits accordés à une entité active du système. Ce nouvel ensemble de droits doit être suffisant pour que l'entité puisse réaliser sa tâche.*

Cette propriété permet de réduire le nombre d'interactions entre les processus privilégiés et ainsi réduire les comportements potentiellement non désirés sur le système.

2.1.2.3 Séparation des privilèges

Le principe de la séparation des privilèges établit que des actions privilégiées (modification de la configuration, mises à jour, etc.) doivent être réalisées par des utilisateurs ou processus différents. Une première définition a été donnée par [Clark et Wilson, 1987]. Ils proposent d'assurer la cohérence des objets du système en faisant une représentation fidèle des objets du système vis-à-vis de la réalité. Or, il n'est cependant pas possible que cette cohérence soit assurée par un système informatique car les systèmes d'information n'ont pas des senseurs leur permettant d'analyser l'environnement extérieur. Ils proposent donc de séparer les opérations en différentes sous-parties et que chaque partie soit exécutée par des utilisateurs différents, en prenant l'exemple d'un processus d'achat, qui est divisé en plusieurs parties réalisées par des personnes différentes : la personne qui fait la demande, celle qui fait la commande, celle qui paye, etc. Il est possible d'assurer la cohérence des objets avec la réalité.

Définition 2.1.7 ***Séparation des privilèges*** *La séparation des privilèges implique que les privilèges soient distribués sur l'ensemble des utilisateurs.*

Dans un système d'exploitation, ce principe est généralement appliqué au niveau des processus système. Les processus privilégiés qui créent des fichiers n'ont généralement pas le droit de les exécuter par la suite. On peut aussi étendre cette définition en l'appliquant directement aux utilisateurs.

2.1.2.4 Non-interférence

Le principe de non-interférence [Focardi et Gorrieri, 2001] implique que plusieurs processus puissent s'exécuter simultanément sans interférer, c'est-à-dire sans modifier la vision des données ou le comportement des autres processus.

Définition 2.1.8 ***Non-interférence*** *La non-interférence est la possibilité pour une entité de s'exécuter sur un système sans être influencée par une autre entité du système. Cette propriété se caractérise par le fait que pour deux ensembles d'objets donnés, les données du premier ensemble ne sont pas affectées par les actions faites sur le second ensemble.*

2.1.3 Discussion

Les propriétés générales de sécurité permettent d'exprimer de manière informelle des objectifs de sécurité précis. Dans le cas d'un système d'exploitation, ces propriétés sont, suivant les situations, soient trop expressives et conduisent à un blocage du système, soient pas assez. Il a donc été nécessaire de définir des propriétés de sécurité adressant des objectifs de sécurité plus spécifiques. Ce sont ces raisons qui ont conduits les auteurs à définir des raffinements aux propriétés générales. Elles sont plus précises et permettent de gérer des cas visant à compromettre la confidentialité ou l'intégrité, voir des deux.

Nous illustrons, dans la figure 2.1, comment se positionnent les propriétés dérivées vis-à-vis des propriétés générales. On peut ainsi noter que la propriété de non-interférence couvre à la fois la disponibilité, l'intégrité et la confidentialité. La propriété de moindre privilège couvre, quant à elle, les propriétés d'intégrité et de confidentialité.

Ces propriétés de sécurité, qu'elles soient générales ou dérivées, sont difficilement applicables à l'ensemble d'un système d'information. En effet, il est difficile d'assurer par exemple, l'isolation total d'un processus au sein d'un système sans en bloquer le fonctionnement. C'est pour cela que nous allons maintenant voir les modèles de protections qui permettant en pratique de garantir la confidentialité ou l'intégrité en spécialisant le contrôle pour certains éléments (processus, ressources) du système.

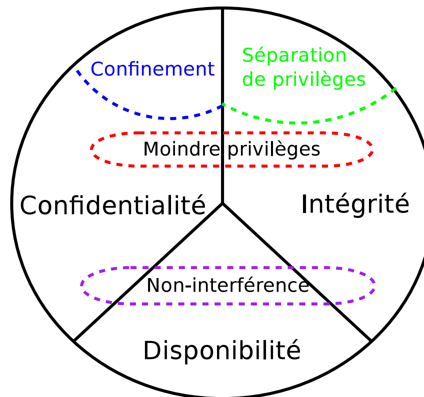

FIGURE 2.1 – Couverture des propriétés dérivées vis-à-vis des propriétés générales de sécurité

2.2 Modèles de contrôle d'accès

La mise en œuvre des propriétés de sécurité que nous venons de définir peut se faire soit en utilisant des mécanismes cryptographiques soit en utilisant des mécanismes de contrôle d'accès. Nous nous intéressons ici exclusivement aux mécanismes de contrôle d'accès des systèmes. Les systèmes d'exploitation se basent sur divers éléments :

— un ensemble de **sujets** : ce sont les entités actives sur le système, essentiellement les processus ;
— un ensemble d'**objets** : ce sont les entités passives du système. Ces entités subissent les interactions effectuées par les sujets. Cet ensemble regroupe toutes les ressources du système (fichiers, socket, etc.).
— un ensemble de **permissions** : ce sont les interactions autorisées des sujets sur les objets (lecture, écriture, etc.) ou entre sujets (envoi d'un signal).

Le contrôle d'accès est l'ensemble des règles qui associent à un sujet un ensemble de permissions sur un objet. Des règles d'accès peuvent aussi exister entre deux sujets.

La manière la plus naturelle de représenter les règles d'accès est d'utiliser une matrice d'accès, comme le montre la table 2.1. Dans cette matrice, les lignes contiennent les sujets, les colonnes correspondent aux objets et l'intersection des deux contient l'ensemble des permissions du sujet sur l'objet.

	Objets			
	$Fichier_1$	$Fichier_2$	$Repertoire_1$	$Sujet_3$
$Sujet_1$	Lecture, écriture	Lecture	Lecture, exécution	Signaux
$Sujet_2$	Lecture		Lecture, écriture, exécution	
$Sujet_3$	Exécution	Ecriture		

TABLE 2.1 – Représentation règles d'accès sous la forme d'une matrice d'accès

2.2.1 Le contrôle d'accès discrétionnaire

Le contrôle d'accès discrétionnaire DAC est le contrôle d'accès que l'on retrouve sur la majorité des systèmes d'exploitation (Linux, Windows et MacOS). Il laisse à la discrétion de l'utilisateur le soin de gérer les accès sur les fichiers qu'il possède. Dans la pratique, les utilisateurs disposent de commandes système leur permettant de définir les accès sur ces fichiers.

2.2.1.1 Lampson

Le premier à avoir formalisé ce modèle de contrôle d'accès est Lampson. Il commence par donner deux définitions : la première concerne les capacités et la seconde concerne les listes de contrôle d'accès (*Access Control List*, ACL). Ces deux définitions sont décrites dans [Lampson, 1969]. [Lampson, 1971] affine ces définitions grâce à la modélisation de la notion de matrice d'accès. Il propose de placer dans une matrice A, l'ensemble D des domaines de protection qui représentent des contextes d'exécution pour les programmes (les sujets) sur les lignes, et en colonnes l'ensemble X des objets (incluant les domaines).

Lampson donne des définitions de chaque terme. Un objet est une "chose" du système qui doit être protégée. Cela peut être, par exemple, les processus, les domaines, les fichiers, etc. Les domaines sont les entités du système qui ont accès aux objets. La propriété essentielle d'un domaine est qu'il a des permissions différentes des autres domaines. De part cette définition, les domaines sont donc aussi des objets. Les capacités (*capabilities*) sont des privilèges accordés aux domaines. Sous Linux, ces privilèges scindent ceux du super-utilisateur en capacité que l'on peut accorder ou interdire à chaque utilisateur.

Définition 2.2.1 ***Liste des capacités*** *Étant donné un domaine $d \in D$, la liste des capacités pour le domaine d est l'ensemble des couples $(o, A[d, o])$, $\forall o \in X$.*

Définition 2.2.2 ***Liste de Contrôle d'Accès*** *Étant donné un objet $o \in X$, la liste de contrôle d'accès pour l'objet o est l'ensemble des couples $(d, A[d, o])$, $\forall d \in D$.*

Lampson ajoute deux nouvelles capacités : le droit `propriétaire` et le droit de copie symbolisé par $\star$. Dans la table 2.2, le droit propriétaire ajouté au $Domaine_1$ l'autorise à contrôler les droits sur le $Fichier_1$ pour l'ensemble des domaines. D'un point de vue de la matrice, il est capable de modifier les droits sur l'ensemble de la colonne. Le droit de copie $\star$, s'applique sur une permission spécifique, autorisant le domaine qui possède cette capacité à recopier la permission sur toute la colonne correspondante.

	Objets			
	$Fichier_1$	$Fichier_2$	$Repertoire_1$	$Process_1$
$Domaine_1$	Lecture, écriture, propriétaire	Lecture	Lecture, exécution	Signaux
$Domaine_2$	Lecture		Lecture, écriture, exécution	
$Domaine_3$	Exécution	*Ecriture		

FIGURE 2.2 – Représentation de Lampson d'une matrice d'accès

Ce modèle a ensuite évolué vers le modèle HRU [Harrison *et al.*, 1976].

2.2.1.2 HRU

Dans le modèle HRU [Harrison *et al.*, 1976], les auteurs modélisent le contrôle d'accès discrétionnaire à partir d'une matrice P, qui représente l'ensemble des droits d'accès des sujets sur les objets. Les sujets peuvent créer des sujets, des objets et modifier les permissions.

Les auteurs du modèle HRU proposent de modéliser le contrôle d'accès de la manière suivante :
- l'ensemble des sujets S et l'ensemble des objets O ;
- l'ensemble des droits génériques R tels que possession, lecture, écriture, exécution ;
- une configuration de protection système repose sur le triplet (S, O, P) ;
- une matrice de contrôle d'accès P ;
- un ensemble fini C de commandes $c_1, ..., c_n$, représente l'ensemble des opérations fournies par le système d'exploitation (création de fichier, modification des droits...) ;

— un ensemble d'opérations élémentaires E : *enter* et *delete* pour l'ajout et la suppression de droits, *create subject* et *create object* pour la création de nouveaux sujets et objets et enfin *destroy subject* et *destroy object* pour la destruction de sujets et objets.

Afin d'étudier le problème de la sûreté d'un système de protection, HRU s'intéresse au transfert de privilège (droit), qui se produit lorsqu'une commande insère un droit particulier r, dans une case de la matrice P où il était précédemment absent. La problématique de sûreté d'un système de protection se définit ainsi : *étant donné une configuration initiale de la politique de sécurité, un système est considéré comme sûr pour un droit r si aucune des commandes de ce système ne provoque le transfert du droit r.*

Les auteurs prouvent que lorsque les commandes ne contiennent qu'une seule action élémentaire, le problème de sûreté est décidable mais l'algorithme de vérification est *NP-Complet*. Ce modèle mono-opérationnel n'est pas du tout représentatif des systèmes courants. Dans le cas général, le problème de la protection d'un système est indécidable. Le problème est néanmoins de taille polynomiale si on retire les opérations de création de sujet et d'objet du modèle de protection.

Dans le cas d'un système d'exploitation, le problème de la vérification d'un contrôle d'accès discrétionnaire est indécidable. Ce résultat prouve qu'il n'est pas possible de garantir des propriétés de sécurité avec un contrôle d'accès discrétionnaire.

2.2.1.3 TAM

Le modèle TAM (*Typed Access Matrix*) exprimé par [Sandhu, 1992], propose une extension du modèle HRU en intégrant la notion de typage fort. Cette notion provient de travaux plus anciens de SPM (*Sandhu's Schematic Protection Model*) [Sandhu, 1988] et se traduit par l'attachement de types de sécurité immuables à tous les sujets et objets du système.

Le modèle HRU est enrichi d'un nouvel ensemble T des types de sécurité. Cet ensemble est un ensemble fini, c'est-à-dire qu'il n'est pas possible de créer de nouveaux types. La gestion des types est prise en compte dans les opérations élémentaires décrites pour le modèle HRU.

L'auteur s'intéresse ensuite à la version monotone du modèle TAM, MTAM (*Monotonic Typed Access Matrix*) qui se caractérise par la suppression des opérations de suppression (droits, sujets et objets). À partir du modèle MTAM, il démontre que le problème de sûreté de fonctionnement est décidable car le graphe de création des sujets et des objets est acyclique. Cependant, la complexité du problème reste NP. C'est pourquoi Sandhu définit le modèle MTAM ternaire, dans lequel toutes les commandes ont au maximum trois arguments. Au prix d'une perte d'expressivité, le problème de sûreté voit sa complexité ramenée à un degré polynomial.

En pratique, ce modèle théorique est difficilement applicable car il revient à ne jamais modifier la politique de contrôle d'accès.

2.2.1.4 DTAM

La modèle DTAM (*Dynamic-Typed Access Matrix*) est exprimé par [Soshi *et al.*, 2004]. Ils proposent une extension du modèle TAM. Le but de ce modèle est d'autoriser les types des objets à être changé de façon dynamique, à la différence du modèle TAM où ils sont statiques. De plus, ce modèle permet de décrire les systèmes de protection non-monotonique pour lesquels la problématique de sûreté est décidable.

Le modèle se base sur le modèle TAM à l'exception que les objets peuvent changer de types de façon dynamique et que l'ensemble des types est un fini. Les auteurs ajoutent donc des commandes supplémentaires aux commandes définies par HRU telles que **change type of subject** et **change type of object**.

Les auteurs du modèle DTAM démontrent que la sûreté du système de protection est décidable en s'appuyant sur le fait que la commande **create** génère des changements irréversibles sur les

types des objets parents. Cela signifie que lorsqu'un domaine crée un objet via la commande **create**, le domaine change aussi de types. De plus, un objet ne peut avoir un type qu'il a possédé auparavant, donc le nombre de changements de types pour un objet est fini puisque le nombre de types est fini. Par conséquence, le problème de sûreté devient décidable dans ce modèle. Il est à noter que dans ce modèle, il est possible que le système atteigne un état dans lequel il ne puisse plus créer de nouveaux objets lorsque tous les objets du système ont utilisés tous les types possibles.

Cela revient à avoir une politique figée où l'on ne peut définir de nouveaux types. Ce modèle théorique est donc aussi difficilement utilisable en pratique.

2.2.2 Le contrôle d'accès obligatoire

Le contrôle d'accès discrétionnaire laisse l'utilisateur final décider des permissions sur les objets qu'il possède. Les différents modèles de contrôle d'accès discrétionnaire ne permettent pas d'avoir un système *sûr*. [Anderson, 1972] détaille le fait qu'il est possible de se protéger des attaques extérieures par l'ajout d'outils, mais qu'il n'est pas possible de se protéger des attaques venant de l'intérieur. Ces attaques peuvent être malicieuses ou accidentelles.

Il est donc nécessaire d'imposer une politique de contrôle d'accès aux utilisateurs du système, en utilisant un modèle de contrôle d'accès obligatoire (MAC, *Mandatory Access Control*). Cette politique ne doit pas pouvoir être modifiée par les utilisateurs finaux. [Anderson, 1980] propose l'utilisation d'un **Moniteur de Référence**. Ce moniteur de référence propose une base de confiance associée à une matrice d'accès qui précise de manière explicite tous les accès autorisés ou refusés au sein du système. De part son caractère non-modifiable par les utilisateurs finaux, le moniteur permet de garantir des propriétés associées à la matrice d'accès.

Cette thèse ayant pour objectifs de permettre aux systèmes d'exploitation de garantir des propriétés de sécurité, nous nous focaliserons uniquement sur les modèles de contrôle d'accès obligatoire proches de nos objectifs. D'autres modèles tels que le modèle de Clark-Wilson, de la muraille de chine, sont décrits dans [Rouzaud-Cornabas, 2010].

2.2.2.1 Bell-LaPadula

Le modèle de Bell-LaPadula (BLP), défini par [Bell et La Padula, 1973], formalise la propriété de confidentialité des données dans les milieux de défense. En plus des notions classiques de sujet, d'objet et de permissions, ce modèle introduit la notion de label. Ce label est un niveau de sécurité, qui va contenir deux types d'identifiant de sécurité. Le premier élément est le niveau hiérarchique, qui renseigne sur le niveau de classification de l'objet ou sur le niveau d'habilitation du sujet. Le second élément est un identifiant de catégories, qui renseigne sur le type de population capable de manipuler les informations comme *public*, *privé* ou *militaire*. Ces identifiants sont indépendants de la hiérarchie de confidentialité.

À partir de ces niveaux de sécurité, il est possible de définir deux règles qui régissent le système en plus des mécanismes de contrôles d'accès classiques.

- **ss-property** : *simple security property*. Cette propriété assure qu'un sujet qui demande un accès en lecture sur un objet du système possède un niveau de sécurité supérieur ou égal à celui de l'objet.
- ***-property** : *star property*. Seuls les transferts d'informations depuis des objets de classification inférieure vers des objets de classification supérieure sont autorisés. Cette règle assure donc la prévention contre la divulgation d'informations.

Le système est modélisé de la façon suivante :

- un ensemble de sujet S ;
- un ensemble d'objets O ;

— une matrice d'accès M ;
— une fonction donnant le niveau f.

On dispose également d'un ensemble de permissions d'accès $A = \{e, r, a, w\}$. Elles sont classées suivant leur capacité d'observation (lecture) et d'altération (écriture) de l'information :

— *execute* : e Ni observation ni altération ;
— *read* : r Observation sans altération ;
— *append* : a Altération sans observation ;
— *write* : w Observation et altération.

On obtient ainsi les deux lois suivantes :

$$\textbf{ss-property} : r \in M[s, o] \rightarrow f(s) \geq f(o)$$
$$\textbf{*-property} : r \in M[s, o_1] \wedge w \in M[s, o_2] \rightarrow f(o_2) \geq f(o_1)$$

FIGURE 2.3 – Lois d'application des propriétés de BLP

Cependant, la propriété ***-property** peut être contournée en exploitant les canaux cachés présents sur le système. Ces canaux cachés échappent complètement au mécanisme de contrôle d'accès présent sur le système. Afin de prévenir ce genre de problème, une version plus restrictive de la politique de BLP utilise les règles suivantes :

— **No Read Up** : cette propriété assure qu'un sujet qui demande un accès en lecture sur un objet du système possède un niveau de sécurité supérieur ou égal à celui de l'objet.
— **No Write Down** : cette propriété assure qu'un sujet qui demande à modifier un objet possède bien un niveau de sécurité inférieur ou égal à celui de l'objet.

Les trois lois sur les niveaux, illustrées par la figure 2.5, s'écrivent ainsi :

$$r \in M[s, o] \rightarrow f(s) \geq f(o)$$
$$a \in M[s, o] \rightarrow f(s) \leq f(o)$$
$$w \in M[s, o] \rightarrow f(s) = f(o)$$

FIGURE 2.4 – Lois d'application des propriétés de BLP-restrictives

Ces lois signifient que :

1. un sujet s a accès en lecture à un objet o si et seulement si son niveau d'habilitation $f(s)$ est supérieur ou égal au niveau de classification $f(o)$ de l'objet ;
2. un sujet s a accès en écriture seul (ajout) à un objet o si et seulement si son niveau d'habilitation est inférieur ou égal au niveau de classification de l'objet ;
3. un sujet s a accès en lecture/écriture à un objet o si et seulement si son niveau d'habilitation est égal au niveau de classification de l'objet.

La figure 2.5 illustre comment se caractérise l'application de ce modèle de protection pour les utilisateurs. Pour qu'un utilisateur puisse accéder à une information, c'est-à-dire puisse la lire, il doit posséder un niveau d'habilitation supérieur ou égal à celui de l'objet. Si un utilisateur veut modifier une information, il doit posséder un niveau d'habilitation inférieur ou égal à celui de l'objet.

Ce modèle est difficilement intégrable dans les systèmes d'exploitation courant. En effet, il est difficile de définir des labels pour certains objets du système qui doivent être partagés entre les différents utilisateurs et le système. On peut ainsi citer les répertoires temporaires. Il existe des mécanismes d'aménagement de ce modèle pour le rendre utilisable. Ces mécanismes (*trusted subjects*) font transiter les objets vers un niveau de sécurité suffisant pour qu'ils puissent être modifiés par les sujets ayant un niveau de sécurité plus élevé. Ainsi la propriété de **No Write Down** n'est pas enfreinte. Cependant, ces mécanismes conduisent à faire monter l'information au

niveau de la confidentialité la plus élevée, ce qui oblige alors à des déclassifications complexes. Actuellement, peu de systèmes d'exploitation adoptent ces approches pour l'ensemble du système même si certaines solutions utilisent la classification sous un autre angle.

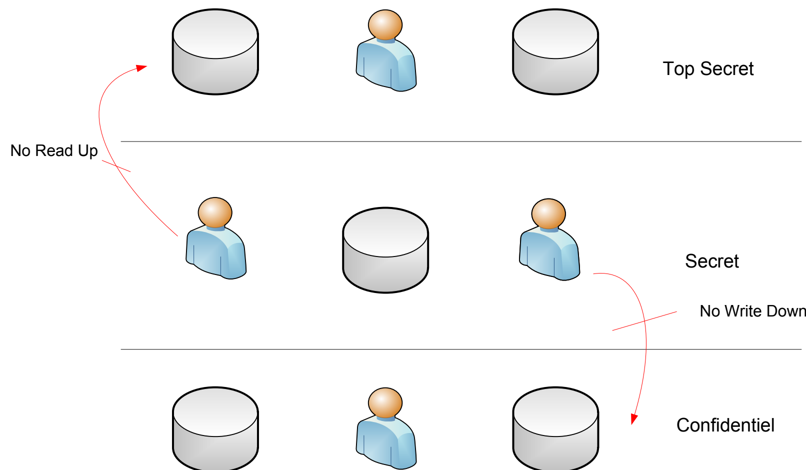

FIGURE 2.5 – Modèle **No Read Up/No Write Down**

2.2.2.2 Biba

Le modèle de BLP traite le problème de la confidentialité des données du système, mais ne propose pas de solution pour protéger l'intégrité de ces données et du système. C'est pour cette raison que Biba a défini un modèle dual à celui de BLP qui formalise les besoins en intégrité d'un système [Biba, 1975]. Tous les sujets et les objets possèdent un niveau d'intégrité. C'est à partir de ce niveau d'intégrité que de nouvelles restrictions sont mises en place sur le système. De manière similaire au modèle de BLP, il propose aussi deux propriétés.

— **No Read Down** : cette propriété assure qu'un sujet qui demande un accès en lecture sur un objet du système possède un niveau d'intégrité inférieur ou égal à celui de l'objet.
— **No Write Up** : cette propriété assure qu'un sujet qui demande un accès en écriture sur un objet du système possède un niveau d'intégrité supérieur ou égal à celui de l'objet.

Les lois s'écrivent donc :

$$r \in M[s,o] \rightarrow f(s) \leq f(o)$$
$$a \in M[s,o] \rightarrow f(s) \geq f(o)$$
$$w \in M[s,o] \rightarrow f(s) = f(o)$$

FIGURE 2.6 – Lois d'application des propriétés de Biba

Ces lois signifient que :

1. un sujet s a accès en lecture à un objet o si et seulement si son niveau d'intégrité $f(s)$ est inférieur ou égal au niveau d'intégrité $f(o)$ de l'objet ;
2. un sujet s a accès en écriture à un objet o si et seulement si son niveau d'intégrité est supérieur ou égal au niveau d'intégrité de l'objet ;
3. un sujet s a accès en lecture/écriture à un objet o si et seulement si son niveau d'intégrité est égal au niveau d'intégrité de l'objet.

Ces règles évitent que se produise un transfert d'information d'un niveau d'intégrité bas vers un niveau d'intégrité haut, ce qui signifierait une compromission de l'intégrité du niveau haut.

La figure 2.7 illustre l'application du modèle de protection de Biba.

Tout comme le modèle proposé par BLP, le modèle de Biba est difficilement intégrable sur un système d'exploitation courant. En effet, pour que le système soit fonctionnel, il est nécessaire de définir des mécanismes modifiant les niveaux d'intégrité de certains sujets du système. Par cette modification de leur niveau d'intégrité, les sujets sont naturellement descendus vers le niveau d'intégrité le plus bas. De plus, le modèle de Biba interdit les accès non autorisés aux ressources mais pas les mauvaises modifications réalisées par des utilisateurs autorisés [Clark et Wilson, 1987].

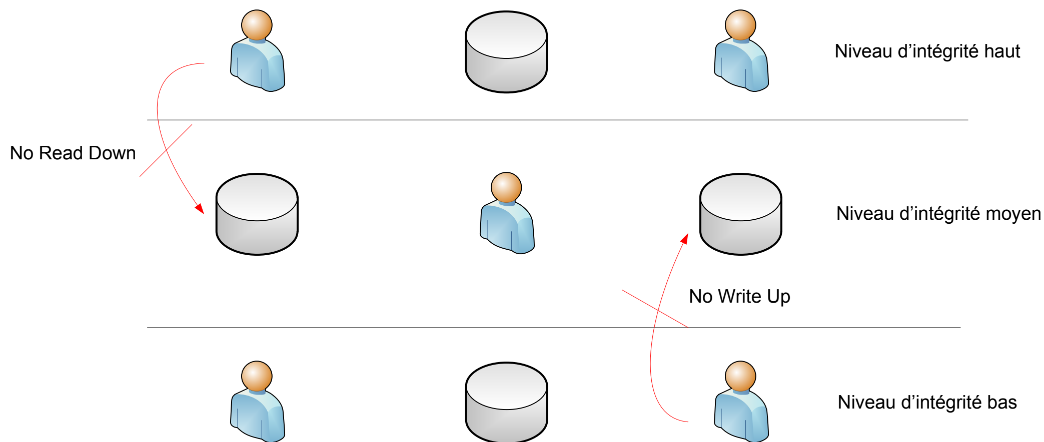

FIGURE 2.7 – Modèle de protection des données : Biba

2.2.2.3 Role Based Access Control

L'administration d'un mécanisme de contrôle d'accès obligatoire est difficile pour un administrateur. Cela implique l'écriture et la gestion d'un grand nombre de règles. L'écriture des politiques de contrôle d'accès peut se révéler extrêmement difficile et coûteuse en termes de temps. Le modèle de contrôle d'accès basé sur les rôles (RBAC) propose une solution pour diminuer cette complexité d'écriture.

Dans la majorité des cas, les permissions d'accès ne sont pas accordées directement en fonction des utilisateurs mais plutôt par rapport à leur activité dans l'entreprise comme le montre [Ferraiolo et Kuhn, 1992]. Ce modèle représente les activités par des rôles.

Par la suite, un modèle plus formel a été défini par [Sandhu *et al.*, 1996], qui se nomme $RBAC_0$. Les utilisateurs peuvent accéder à un ensemble de rôles. Puis, ce sont ces rôles qui sont associés à des permissions dans le but d'effectuer des actions sur le système d'exploitation. Enfin, l'utilisateur exerce son activité dans le cadre de sessions, durant lesquelles il peut activer un sous-ensemble des rôles auxquels il appartient. Ainsi, les politiques de contrôle d'accès s'établissent à partir des autorisations données aux rôles plutôt qu'aux utilisateurs.

A partir de ce modèle, une notion de hiérarchie des rôles a été mise en place dans [Sandhu *et al.*, 1996], modèle nommé $RBAC_1$. Cette hiérarchie permet d'avoir un héritage des permissions dans les rôles, suivant la place de la personne dans l'entreprise. Comme le montre la figure 2.8, on retrouve le *junior* en bas du diagramme et il ne possède que peu de permissions. Alors que le *senior* est lui placé en haut du diagramme et hérite de toutes les permissions des rôles intermédiaires entre le *junior* et le *senior*.

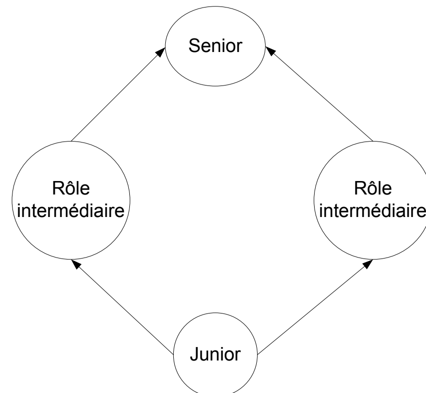

FIGURE 2.8 – Schéma d'héritage du modèle $RBAC_1$

À partir du modèle $RBAC_0$, [Sandhu *et al.*, 1996] définit un troisième modèle nommé $RBAC_2$ qui introduit le support des contraintes. L'objectif de ce support est de modéliser les contraintes des rôles qui peuvent être totalement disjoints.

Les types de contraintes supportés dans $RBAC_2$ sont généralement des considérations simples. Elles portent soit sur l'attribution des rôles aux utilisateurs, soit sur l'attribution des permissions aux rôles. Les principaux types de contraintes sont les suivants :

— *rôles mutuellement exclusifs* : un utilisateur ne peut être affecté qu'à un seul rôle dans un ensemble de rôles mutuellement exclusifs. Il s'agit d'une propriété de séparation des privilèges ;

— *cardinalité* : contrainte sur le nombre maximal d'utilisateurs affectés à un rôle ;

— *rôles prérequis* : contrainte qui impose un prérequis pour l'ajout d'un utilisateur dans un rôle. Par exemple, un utilisateur doit faire partie d'un certain rôle pour pouvoir être ajouté dans le nouveau.

Le modèle $RBAC_3$ [Sandhu *et al.*, 1996] permet de consolider les trois modèles précédents. Il supporte donc à la fois les hiérarchies de rôles et les contraintes sur l'assignation des rôles et des permissions. L'objectif de ce dernier modèle est de résoudre les problèmes soulevés par la cohabitation de ces deux concepts : support des contraintes sur la hiérarchie de rôles, résolution de conflits entre l'héritage et les contraintes (par exemple quand un rôle hérite de deux rôles mutuellement exclusifs). En pratique, les systèmes d'exploitation utilisent la notion de rôle de façon simplifiée. L'utilité est alors une factorisation d'un ensemble de règles pour différents utilisateurs en une règle associée à un rôle. L'objectif est de regrouper les utilisateurs dans des rôles pour ensuite établir les règles par rapport à ces rôles et non plus par rapport aux utilisateurs.

2.2.2.4 Domain and Type Enforcement

Le modèle de protection *Domain and Type Enforcement*(DTE) [Boebert et Kain, 1985] est un modèle basé sur une abstraction des ressources du système, créé spécifiquement pour écrire des politiques MAC. Ce modèle ne s'appuie pas sur les notions de sujets et d'objets mais sur celles de *domain* et de *type* pour distinguer les entités actives et passives du système. Un type est une chaîne de caractères qui caractérise une ressource du système.

La figure 2.9 montre que ce modèle autorise les interactions d'un domaine sur un type, mais aussi les interactions entre les domaines.

Le modèle de protection établit par DTE diffère de BLP ou Biba. Alors que ces deux modèles implantent une des propriétés générales de sécurité, le modèle DTE répond à la problématique de

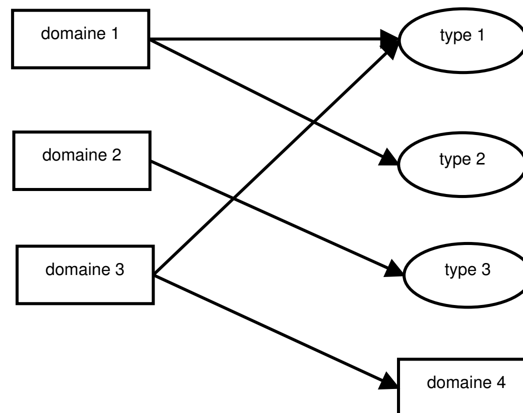

FIGURE 2.9 – Schéma du modèle DTE

contrôle des activités des processus. En effet, ce modèle permet de définir précisément les accès autorisés aux domaines.

Le modèle DTE permet donc de répondre à deux objectifs de sécurité :

— restreindre les ressources accessibles par un processus : ce modèle s'applique à l'ensemble du système, même aux processus tournant avec des privilèges élevés. C'est aussi une approche de moindre privilège telle que définie précédemment ;
— établir un contrôle strict des processus ayant accès aux ressources jugées sensibles et empêcher tout programme non autorisé à y accéder.

Les entités actives du système s'exécutent dans un domaine précis alors que les objets du système possèdent tous un type. Ainsi, les lois d'accès opèrent entre les domaines et les types. Ce modèle se base aussi sur la notion de matrice d'accès. Le modèle DTE détermine trois tables :

— la table de typages, qui associe un type à chaque objet du système ;
— la table de définition des domaines qui détermine les droits d'accès (lecture, écriture, exécution, ajout, suppression) de chaque domaine sur les différents types ;
— la table d'interaction des domaines qui établit les droits d'accès entre domaines (création, destruction, envoi de signal).

La table de définitions des domaines détermine aussi le type d'un programme. L'exécution d'un programme binaire fait entrer le processus dans un domaine précis. Ce modèle a le mérite d'être facile à mettre en œuvre et correspond à la notion de moniteur de référence et de confiance. Il permet de garantir un large ensemble de cas particuliers de confidentialité et d'intégrité. Le principal problème concerne les propriétés incluses dans la politique qui ne sont pas définies au moyen d'un formalisme de haut niveau, ce qui peut rendre leur écriture difficile.

2.2.3 Discussion

Nous venons de détailler dans les deux premières parties de cet état de l'art les moyens d'expression d'objectifs de sécurité et des modèles d'application. Les propriétés de sécurité exprimées ici sont des synthèses des définitions que nous pouvons retrouver dans les documents [ITSEC, 1991, TCSEC, 1985].

Les modèles de contrôle d'accès que nous avons présentés offrent des modélisations des objectifs de sécurité. La première remarque que nous pouvons faire est qu'aucun modèle de protection n'implante toutes les propriétés de sécurité que nous avons évoquées.

Dans les systèmes d'exploitation courants tels que Linux, MacOs et Windows, nous retrouvons le DAC. Ce modèle de protection ne permet ni de protéger le système, ni les données qu'il héberge. [Harrison *et al.*, 1976] a démontré que la problématique de sûreté de fonctionnement pour un sys-

tème se reposant sur le contrôle d'accès discrétionnaire n'était décidable qu'au prix de grandes simplifications qui ne correspondent pas à la réalité des systèmes. Dans le cas des systèmes courants, le problème de sûreté n'est donc pas décidable.

Les modèles de contrôle d'accès obligatoire offrent des réponses à la problématique de garantie par le système d'exploitation de propriétés de sécurité. La plupart de ces modèles ont été établis pour répondre à des problématiques du monde de défense, c'est-à-dire extrêmement spécifiques et contraignantes. L'implantation réelle de ces modèles dans des systèmes existent, par exemple le modèle de BLP dans MULTICS, on le trouve également dans certaines versions de Solaris, HPUX ou autres systèmes UNIX. Cependant, pour les systèmes d'exploitation courants, ils sont difficilement intégrables car il est nécessaire de créer des exceptions qui mettent le modèle de protection à mal. De plus, ces systèmes ne traitent que d'un cas particuliers des propriétés de sécurité qui est la confidentialité.

Le modèle RBAC offre peu d'application pratique en termes de systèmes d'exploitation et sert essentiellement à la factorisation des règles de contrôle d'accès. Enfin, le modèle DTE est le plus flexible. S'il n'est pas dédié à une propriété particulière, il permet d'écrire des politiques incluant un large ensemble de propriétés de confidentialité et d'intégrité entre les processus et les ressources, tout en minimisant les privilèges des processus.

2.3 Implantation des mécanismes de contrôle d'accès

Dans cette section, nous allons nous intéresser à l'implantation des mécanismes de contrôle d'accès obligatoire présents à la fois sous Linux et sous Windows. Il existe d'autres implantations présentes sur les systèmes Android, Solaris, etc, mais ces systèmes sortent du cadre de notre étude.

Sur les systèmes d'exploitation Linux, il existe un plus grand nombre d'implantations de mécanismes de contrôle d'accès obligatoire que sous Windows car son code source est ouvert. Comme l'objet de cette étude n'est pas de décrire l'implantation de chaque mécanisme présent sous Linux, nous nous restreindrons à quatre implantations de références. Nous avons choisi de décrire SELinux, grsecurity, Tomoyo et PIGA. SELinux et grsecurity font partis des mécanismes les plus anciens et les plus matures sur les systèmes Linux. Leur intégration au sein du système d'exploitation est aussi différente. Nous détaillons aussi Tomoyo et PIGA qui sont deux mécanismes différents des deux premiers et qui ne réalisent pas le même type de contrôle. Une description des projets MEDUSA, RSBAC et LIDS, offrant d'autres implantations de contrôle d'accès obligatoire, est présente dans la thèse [Blanc, 2006].

Sur les systèmes d'exploitation Windows, nous nous attacherons à détailler le mécanisme de contrôle d'intégrité (*Mandatory Integrity Control*, MIC) mis en place par Microsoft à partir des systèmes d'exploitation Windows Vista. Nous avons aussi choisi de décrire PRECIP, qui est une implantation restrictive du modèle de BLP ainsi que Core Force, le premier mécanisme MAC apparu sous Windows XP.

2.3.1 Linux

Dans cette partie, nous allons traiter des mécanismes de contrôle d'accès obligatoire sur les systèmes Linux.

2.3.1.1 SELinux

SELinux(*Security-Enhanced Linux*) est un mécanisme de contrôle d'accès obligatoire créé par la NSA et intégré nativement dans les noyaux Linux. Il est proposé sous la forme de module

de sécurité s'interfaçant avec les crochets de sécurité (*Security Hooks*) du noyau (*Linux Security Module* (LSM) [Wright *et al.*, 2002]).

Basé sur le modèle de DTE, SELinux implante les propriétés de moindre privilège et de confinement des processus pour garantir la confidentialité et l'intégrité. Son architecture est basée sur FLASK [Spencer *et al.*, 1998], une architecture spécifiquement créée pour SELinux, qui offre une couche d'abstraction entre le système et le mécanisme de contrôle d'accès sous-jacent, transformant les ressources du système en contexte de sécurité. Ainsi les ressources du système ne sont pas identifiées par leur nom ou par leur chemin, mais par des contextes de sécurité.

L'écriture d'une politique SELinux étant complexe, SELinux utilise également le modèle RBAC pour réduire la taille de la politique. Ainsi, les utilisateurs sont associés à des rôles qui peuvent ensuite accéder à des domaines, et ce sont les domaines qui agissent sur les types.

Modèle général

Chaque utilisateur du système (utilisateur standard, mais aussi administrateur) est associé à une identité SELinux unique. Cette identité SELinux peut être partagée par plusieurs utilisateurs. L'identité SELinux est associée à un rôle par défaut, mais elle peut aussi accéder à un ensemble de rôles. Les utilisateurs ont la possibilité de changer de rôle sans changer de session ou de compte utilisateur. Les rôles accèdent ensuite aux domaines.

Chaque ressource du système possède un contexte de sécurité qui la caractérise. Le contexte de sécurité est considéré comme une couche d'abstraction entre les ressources et le mécanisme de contrôle d'accès. Il contient les informations nécessaires au modèle DTE (domaines ou types).

Contextes de sécurité

L'architecture FLASK crée une couche d'abstraction pour la représentation du système qui lui permet de n'être qu'une interface entre le système d'exploitation et le mécanisme de contrôle d'accès sous-jacent. Cette abstraction permet l'association de contextes de sécurité avec les ressources du système.

Les contextes de sécurité sont définis par le **serveur de sécurité** qui constitue un moniteur de référence au sens d'Anderson. Pour SELinux, un contexte de sécurité est constitué de plusieurs attributs :

- une identité qui est liée aux identités des utilisateurs Linux présents sur le système. Ainsi, plusieurs utilisateurs standards peuvent avoir la même identité SELinux. Cette identité SELinux a la particularité de ne pas pouvoir être modifiée aux cours des interactions réalisées. Seuls certains programmes d'authentification ont la capacité de pouvoir changer les identités SELinux ;
- un rôle : un utilisateur SELinux a accès à un ensemble de rôles. Ce sont ces rôles qui déterminent les interactions autorisées pour l'utilisateur ;
- un domaine, qui est le domaine d'exécution du processus ;
- un type, qui est le type d'un objet du système autre qu'un processus ;
- une catégorie : cet attribut est la représentation de l'implantation du modèle *Multi-Categorie Security* de SELinux. Il permet de définir des conteneurs pour les utilisateurs dans le but de les isoler les uns des autres. Les catégories permettent d'isoler des populations ayant la même identité SELinux. Il est possible d'avoir plusieurs catégories pour un même contexte de sécurité.
- un niveau d'habilitation : cet attribut est l'application du modèle de BLP. Les niveaux d'habilitation peuvent être une valeur unique ou une plage de niveaux. C'est l'implantation du modèle *Multi-Level Security* de SELinux.

Un contexte de sécurité est représenté au niveau du système de fichiers par une chaîne de caractères de la manière suivante : `id:rôle:type:catégories:sensibilités`.

Un utilisateur possède une identité SELinux de part son identité standard Linux. Cette identité SELinux lui permet d'accéder à certains nombres de rôles SELinux. Ces rôles peuvent accéder à des domaines. Les domaines peuvent à leur tour accéder à d'autres domaines ou à des types. Ce modèle est illustré par la figure 2.10.

FIGURE 2.10 – Implantation des modèles RBAC et DTE par SELinux

Pour les objets du système, les contextes de sécurité sont stockés dans le système de fichiers au niveau des attributs étendus.

Politique et règle de contrôle d'accès

Les domaines effectuent des interactions sur les objets du système qui possèdent tous un type. Une interaction se caractérise par la règle suivante $D - (P) - > T$, ou D est le domaine, P les permissions et T le type. Le cas spécifique d'une interaction entre deux domaines se traduit de manière similaire $D_1 - (P) - > D_2$.

La politique de contrôle d'accès de SELinux se divise en deux. D'un côté, nous trouvons les règles faisant la correspondance entre les ressources du système et leur contexte de sécurité spécifique. De l'autre, nous trouvons les règles d'accès et de transition.

Les règles d'accès définissent de manière explicite les interactions autorisées par des domaines sur des types du système. Les règles de transition sont les règles qui autorisent la création de nouveau domaine à partir de type spécifique.

La définition d'une règle d'accès dans la politique SELinux se fait donc de la manière suivante 2.1.

```
1 allow linpack_t etc_t:file { read open getattr };
```

Listing 2.1 – Règle d'accès dans une politique SELinux

Cette règle autorise (*allow*) le domaine *linpack_t* à lire, ouvrir et récupérer les attributs des fichiers *file { read open getattr }* ayant pour type *etc_t*. Concrètement, cette règle autorise un processus ayant pour domaine *linpack_t* à aller lire des fichiers de configurations présents dans `/etc`.

Pour autoriser le rôle *staff_r* à accéder au domaine `linpack_t`, il suffit d'ajouter la règle 2.2.

```
1 role staff_r types linpack_t;
```

Listing 2.2 – Règle d'autorisation pour un rôle pour accéder à un type dans une politique SELinux

Le dernier type de règle concerne les transitions. Ce sont des opérations spécifiques qui surviennent lors de la création d'un nouveau domaine. La définition d'une règle de transition se définit de la manière suivante 2.3 :

```
type_transition staff_t mozilla_exec_t:process mozilla_t;
```

Listing 2.3 – Règle de transition dans une politique SELinux

Cette règle autorise le domaine *staff_t* à faire transiter un type *mozilla_exec_t* vers le domaine *mozilla_t*. Cette règle est par exemple, appliquée lorsqu'un utilisateur du staff exécute le programme *Firefox*.

2.3.1.2 grsecurity

grsecurity [Spengler, 2002] est un mécanisme de contrôle d'accès obligatoire présent sur les systèmes d'exploitation Linux. À la différence de SELinux, il n'est pas inclus dans la version courante du noyau car il n'utilise pas les LSM comme le préconise les développeurs du noyau. Il est donc disponible sous la forme d'un correctif noyau. En plus d'un mécanisme de contrôle d'accès, grsecurity fournit un second correctif noyau nommé *PaX*, qui offre des protections contre les attaques de type *buffer overflow*[Team, 2012].

Modèle général

Comme pour SELinux, grsecurity permet de définir des ensembles de règles puis d'affecter ces ensembles à des rôles. En effet, sans cette notion de rôles, tous les utilisateurs auraient les mêmes droits. Un utilisateur peut changer de rôle au sein d'une même session.

À la différence de SELinux qui associe un contexte de sécurité à chaque ressource du système, grsecurity identifie les ressources en utilisant leur chemin complet dans l'arborescence du système de fichiers. Ce modèle de protection se nomme *Path-Based Access Control* (PBAC). Les interactions sont représentées par les ACL définies sur les objets pour chaque sujet. Par défaut, un nouveau processus hérite des permissions de son père, à part s'il possède un jeu de permissions spécifiques.

Politique et règle de contrôle d'accès

La politique de contrôle d'accès de grsecurity s'appuie donc sur les chemins complets des sujets (binaires exécutés correspondant à l'application) et des objets pour établir les règles de contrôle d'accès. Chaque règle associe à un sujet, un objet ainsi que les ACL autorisées sur cet objet. grsecurity permet aussi de gérer les *capabilities POSIX* [Alexander Kjeldaas, 1998] ainsi que les quantités de ressources utilisables [Spender, 2003].

Par défaut, chaque sujet et objet hérite des droits de son père. Ainsi, un fichier créé héritera des permissions du répertoire dans lequel il est. Il est naturellement possible de briser cet héritage pour définir des permissions spécifiques à ces nouveaux objets.

Un règle d'accès dans une politique grsecurity se définit de la manière suivante 2.4.

```
subject /usr/bin/sudo
      /dev/log rw
```

Listing 2.4 – Règle d'accès dans une politique grsecurity

Cette règle autorise le sujet ayant pour chemin complet `/usr/bin/sudo` à lire et écrire (`rw`) l'objet ayant pour chemin `/dev/log`.

grsecurity est aussi capable de gérer les capacités des programmes. Il est donc possible d'ajouter ou d'enlever des capacités aux programmes, comme l'illustre le listing 2.5.

```
1 subject /sbin/syslog-ng
2       +CAP_SYS_ADMIN
3
4 subject /usr/sbin/sshd dpo
5       -CAP_ALL
6       +CAP_CHOWN
```

Listing 2.5 – Règle de modification de *capabilities* dans une politique grsecurity

2.3.1.3 Tomoyo

Tomoyo [Corporation, 2003, Takeda, 2009] est un mécanisme de contrôle d'accès obligatoire apparu en 2003. Depuis la version 2 de Tomoyo sortie en 2007, il utilise les LSM pour réaliser ses contrôles.

Modèle général

Tomoyo reprend la notion de domaine énoncée par [Lampson, 1971], c'est-à-dire qu'un domaine regroupe un ensemble d'entités actives du système qui ont les mêmes capacités. Ainsi Tomoyo construit un domaine en gardant traces des précédents processus impliqués dans la création du processus, qui réalise l'action. Par conséquent, Tomoyo n'utilise pas de label ou de contexte de sécurité pour représenter les ressources du système, mais il se base sur les chemins complets des sujets et des objets, donc sur le modèle PBAC.

Le listing 2.6 donne un exemple de domaine.

```
1 <kernel> /usr/sbin/sshd /bin/bash /usr/bin/man /bin/sh
```

Listing 2.6 – Définition d'un domaine dans une politique Tomoyo

À chaque domaine est associé un ensemble de sujet et d'objet identifiés par leurs chemins complets. La gestion des accès se fait grâce au triplet (r,w,x) sur les objets. Lorsqu'un processus exécute un fichier binaire créant un nouveau processus, cela entraîne une transition vers un nouveau domaine.

L'implantation de Tomoyo permet de représenter, sous la forme d'un graphe, l'évolution de chaque domaine en cours sur le système. À la différence des mécanismes comme SELinux et grsecurity, il est capable de gérer les connexions réseaux et l'administrateur peut donc définir au niveau de la politique de contrôle d'accès, les plages d'adresses IP autorisées pour un domaine donné.

Politique et règles de contrôle d'accès

La politique de contrôle d'accès de Tomoyo s'appuie donc sur deux éléments : les chemins complets des ressources du système et la définition des domaines. La politique associe, à chaque domaine, un ensemble de fichiers binaires que le domaine a le droit d'exécuter dans le but de créer un nouveau domaine. Elle associe aussi un ensemble d'objets (fichier, socket, etc.) auxquels sont associés le triplet de permissions (r,w,x).

La politique se base sur les droits (r,w,x) présents sous Linux. Chaque permission possède une valeur numérique unique. Il est possible d'ajouter ces valeurs pour combiner plusieurs permissions. Ainsi, on fait correspondre au triplet (r,w,x) le triplet (4,2,1). Dans cette représentation, r a pour valeur 4, w 2 et x 1. Pour donner le droit à un domaine d'écriture et de lecture sur un objet, il suffit de donner la valeur $4 + 2 = 6$ sur l'objet pour le domaine concerné.

Le listing 2.7 montre un exemple de politique pour Tomoyo.

```
1 <kernel> /sbin/mingetty /bin/login
2 1 /bin/bash
3 4 /etc/passwd
4 4 /etc/shadow
```

Listing 2.7 – Règle d'accès dans une politique Tomoyo

Cette politique correspond à un domaine représenté par la trace `<kernel> /sbin/mingetty /bin/login`. Ce domaine est autorisé à exécuter (1) le fichier `/bin/bash` et il est autorisé à lire les fichiers `/etc/passwd` et `/etc/shadow`.

Lorsque le domaine exécutera le fichier `/bin/bash` un nouveau domaine sera créé. Ce domaine sera composé des éléments suivants : `<kernel> /sbin/mingetty /bin/login /bin/bash`.

Tomoyo étend les ACL par l'utilisation de mots clés dans sa politique de contrôle d'accès. Ainsi, il est possible, grâce à des expressions régulières, d'autoriser un domaine à accéder à une ressource qui n'est pas encore présente sur le système. L'extension des ACL se traduit aussi par des permissions supplémentaires telles que la possibilité de supprimer une ressource. Le listing 2.8 montre un exemple d'utilisation de l'extension des ACL. Ainsi le langage de configuration supporte à la fois les valeurs numériques des droits classiques mais aussi des mots clés permettant une lecture plus simple des politiques.

```
1 <kernel> /usr/sbin/httpd
2 allow_read /var/www/html/\*
3 allow_read /etc/httpd/\*.conf
4 allow_read /usr/lib/httpd/modules/\*.so
5 allow_write /var/log/httpd/\*_log
6 allow_create /var/run/httpd.pid
7 allow_unlink /var/run/httpd.pid
```

Listing 2.8 – Règle d'accès dans une politique Tomoyo

La politique offre aussi la possibilité de gérer des capacités spécifiques aux domaines. Ainsi, il est possible de spécifier, par exemple, les plages d'IP sur lesquelles le domaine est autorisé à se connecter comme le montre le listing 2.9.

```
1 allow_network
2   TCP accept
3   10.0.0.0-10.255.255.255
4   1024-65535
```

Listing 2.9 – Règle d'accès dans une politique Tomoyo

2.3.1.4 Policy Interaction Graph Analysis

PIGA [Briffaut, 2007] est un mécanisme de contrôle d'accès obligatoire proposé sous deux formes :

- un correctif noyau qui est le *Policy Enforcement Point* (PEP), c'est-à-dire le point d'application de la politique ;
- une implémentation d'un moniteur de référence, qui est donc le *Policy Decision Point* (PDP), c'est-à-dire le moniteur qui prend la décision.

À la différence des mécanismes précédemment présentés comme SELinux et grsecurity, PIGA est capable de gérer à la fois les interactions entre un sujet et un objet ou entre deux sujets, mais aussi des compositions d'interactions (flux entre plusieurs interactions, enchaînement temporel entre les interactions, opérateurs logiques entre interactions, etc.).

PIGA se base sur un langage de spécification de propriétés de sécurité (*Security Property Language*, SPL) de haut niveau, en comparaison de DTE, qui offre la possibilité d'exprimer des propriétés de sécurité 2.1. Ces propriétés de sécurité sont par la suite garanties par le moniteur PIGA.

Modèle général

Dans [Briffaut *et al.*, 2009], les auteurs détaillent le langage d'expression des propriétés de sécurité (SPL). Ce langage indépendant du mécanisme de contrôle d'accès permet d'exprimer des propriétés combinant plusieurs types de flux. La représentation du système utilise les contextes de sécurité définis par SELinux. Il distingue donc l'ensemble des contextes sujets SCs et l'ensemble des contextes objets SCo. Lorsqu'une interaction est effectuée sur le système, il peut y avoir deux effets : un transfert d'information ou une transition.

Le transfert d'information se fait du contexte sujet vers le contexte objet lorsque l'opération associée est une **écriture**. Il se fait dans le sens inverse lors d'une opération de lecture. Une transition est un changement de domaine.

À partir de ces éléments, les auteurs donnent une définition de la dépendance causale entre deux interactions. Ainsi, deux interactions sont causalement dépendantes si :

— elle partage un même contexte de sécurité ;
— la première interaction intervient avant la fin de la seconde ;
— la première interaction modifie l'état du contexte de sécurité partagé ;
— la seconde interaction modifie l'état du contexte de sécurité final.

Cette définition permet ensuite de formaliser les notions d'interactions et de séquences. Une interaction directe, notée $>$, est une opération élémentaire (lecture, écriture, etc.), qui induit un transfert d'information direct entre un processus et une ressource. Une séquence, notée $>>$ correspond à une suite d'interactions causalement dépendantes et induit un flux d'information indirect mettant en jeu au moins 3 contextes, le second servant d'intermédiaire.

Grâce à ces opérateurs de base, les auteurs peuvent exprimer des propriétés de sécurité comme la confidentialité et l'intégrité dans un langage de haut niveau. Ces propriétés de sécurité peuvent être exprimées en utilisant la notion d'interaction, c'est-à-dire de flux d'information direct, ou de séquence, c'est-à-dire de flux d'information indirect, de suite temporelle, d'expression logique, etc.

PIGA est proposé sous la forme de plusieurs outils : PIGA-CC, qui énumère les activités autorisées par une politique MAC contrôlant les interactions directes qui violent les propriétés PIGA exprimées en SPL, PIGA-UM, qui est le moteur de prise de décision et PIGA-Kernel, qui est le hook noyau interceptant les accès autorisés par SELinux.

PIGA-CC, dont le fonctionnement est illustré par la figure 2.11 prend en entrée les propriétés de sécurité écrites en SPL et la politique de contrôle d'accès directe MAC (par exemple : une politique SELinux). Il recherche, dans la politique de contrôle d'accès directe, les activités susceptibles de contourner les propriétés de sécurité. Cette opération se fait en mode hors ligne, c'est-à-dire avant que PIGA ne soit déployé sur un système. Les activités illégitimes calculées sont ensuite utilisées par PIGA-UM. Lorsque PIGA-UM reconstruit l'activité du système, il va vérifier que cette activité ne correspond pas à une activité illégitime.

La partie prise de décision s'effectue en espace utilisateur par PIGA-UM. C'est l'implantation du moniteur de référence définie par [Anderson, 1972]. À partir de chaque interaction capturée en espace noyau, il reconstruit l'activité du système. Cette reconstruction lui permet de vérifier la validité des interactions vis-à-vis des activités qu'il a pré-calculées. Lors de ce calcul, PIGA génère les activités réelles susceptibles de violer les propriétés de sécurité. Par conséquent, si l'interaction

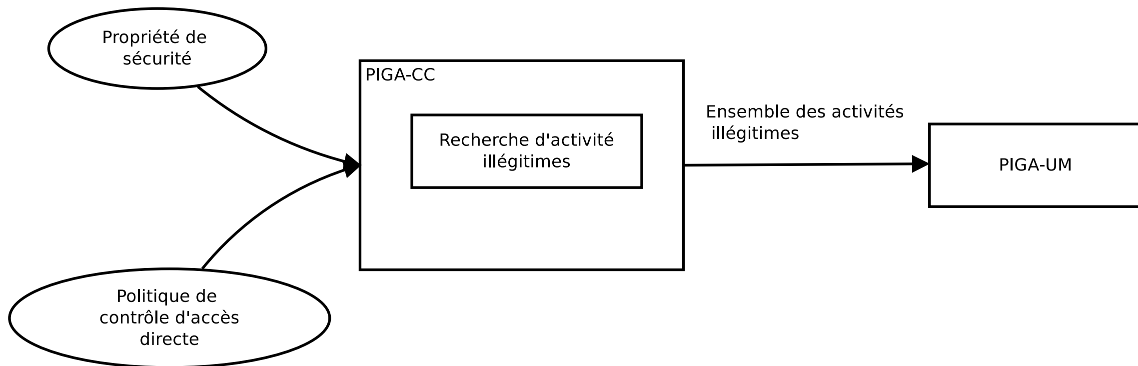

FIGURE 2.11 – Fonctionnement de PIGA-CC

courante fait partie d'une activité réelle qui correspond à une activité illégitime, cette interaction (appel système) est bloquée.

PIGA-Kernel permet de détourner le flux d'exécution des interactions directes. PIGA-Kernel transmet les interactions autorisées à PIGA-UM afin qu'il contrôle les activités correspondant aux scénarios complets.

Lorsqu'un processus effectue un appel système, il est intercepté par SELinux. Si l'appel est autorisé au niveau de SELinux, alors PIGA-Kernel va l'intercepter. Il va ensuite l'envoyer à PIGA-UM. PIGA-UM va ajouter cet appel pour reconstruire l'activité du système, puis il va rechercher dans les signatures qui ont été pré-calculées si cette interaction ne fait pas partie d'une activité illégitime. Si c'est le cas, PIGA-UM a deux possibilités : soit il est en mode en détection et dans ce cas, il se contente de lever une alerte de sécurité sans que l'interaction ne soit bloquée, soit il est en mode protection et par conséquence, l'interaction illégitime échouera c'est-à-dire que le scénario malveillant ne se terminera pas.

2.3.1.5 Politique et règle de contrôle d'accès

Le listing 2.10 montre un exemple de propriété de sécurité exprimée en SPL qui garantit la confidentialité. Ainsi, la propriété 2.10 s'interprète ainsi : à partir de deux contextes de sécurité cibles, $sc1$ qui le contexte sujet et $sc2$ qui est le contexte objet, PIGA va contrôler qu'il n'existe aucun flux d'information indirect $>>$ partant de $sc2$.

```
1 define confidentiality( $sc1 IN SCS, $sc2 IN SCO ) [
2      SA { $sc2 >> $sc1 },
3        { not(exist()) };
4 ];
```

Listing 2.10 – Propriété de confidentialité exprimée en *SPL*

Les résultats fournis par PIGA sont dans ce cas des activités autorisées par la politique de contrôle d'accès directe qui conduisent à un flux indirect (en effet, dans la pratique, les flux directs sont déjà traités par le MAC, comme SELinux). On voit que PIGA améliore la sécurité par rapport à SELinux et permet de formaliser des propriétés de sécurité, ce que ne permet pas SELinux qui est de plus bas niveau.

2.3.2 Windows

Dans cette section, nous allons maintenant détailler des implantations de mécanismes de contrôle d'accès obligatoire pour les systèmes d'exploitation Windows.

2.3.2.1 Mandatory Integrity Control

MIC (*Mandatory Integrity Control*) est un mécanisme de contrôle d'accès obligatoire breveté par Microsoft [Richard Ward, 2006]. Ce modèle, plus récent que les implantations de SELinux et de grsecurity, a été mis en place à partir des systèmes d'exploitation Windows Vista.

À la différence des implantations présentes sous Linux, qui doivent être activées pour SE-Linux, et où il est nécessaire de modifier le noyau pour grsecurity, l'implantation de Microsoft est imposée. Son principal objectif est de protéger le système des attaques, qu'elles viennent de l'extérieur ou de l'intérieur.

Modèle général

Le modèle de protection s'inspire des modèles de Biba et de BLP que nous avons détaillé dans la section 2.2.2.2. Chaque entité du système, que ce soit les sujets ou les objets, possède un niveau d'intégrité [Microsoft, 2013]. Pour qu'un sujet puisse modifier un objet, il doit nécessairement posséder un niveau d'intégrité supérieur ou égal à celui de l'objet.

Cependant, nous avons vu que le modèle de protection défini par Biba était difficilement applicable à un système d'exploitation. Pour résoudre ce problème, Microsoft n'offre que la propriété de *non modification des objets* ayant une intégrité supérieure (*No Write Up*).

Politique et règle de contrôle d'accès

Par défaut, il existe cinq niveaux d'intégrité.

— le niveau de non confiance (*Untrusted*) qui concerne les processus appartenant aux sessions invitées et qui ne peuvent faire aucune interaction privilégiée sur le système ;
— le niveau d'intégrité bas est utilisé pour les processus ayant une forte exposition aux attaques et qui possèdent souvent des failles de sécurité permettant d'entrer sur le système ;
— le niveau d'intégrité moyen qui est le niveau d'intégrité par défaut de l'utilisateur qui s'est connecté. C'est aussi le niveau d'intégrité sous lequel tourne la majorité des programmes lancés par l'utilisateur ;
— le niveau d'intégrité haut, qui est réservé à l'administrateur du système ;
— enfin, le niveau d'intégrité système réservé au système.

Tout comme le définit le modèle basé sur les niveaux d'intégrité de Biba 2.2.2.2, les sujets ayant une intégrité inférieure à celle de l'objet ne peuvent pas le modifier (*No Write Up* qui s'applique spécifiquement aux objets du système). Mais à la différence du modèle théorique, la politique de contrôle d'accès sous Windows n'empêche pas un sujet ayant une intégrité plus élevée que l'objet de le lire (principe du *No Read Down*).

Les lois s'écrivent de la façon suivante :

$$a \in M[s,o] \rightarrow f(s) \geq f(o)$$
$$w \in M[s,o] \rightarrow f(s) = f(o)$$

FIGURE 2.12 – Lois d'application des propriétés du MIC

Pour renforcer son implantation, Microsoft a mis en place, en plus des niveaux d'intégrité, des labels s'appliquant sur les sujets du système. Ces labels agissent en complément des niveaux d'intégrité. Ils permettent de renforcer le contrôle d'accès présent sur les systèmes Windows.

Par défaut, trois labels de sécurité sont définis :

— *No Read Up* : les sujets ayant une intégrité strictement inférieure à celle du sujet cible ne peuvent pas lire le contenu du sujet. Ce label empêche les processus d'intégrité basse d'ob-

server les processus ayant une intégrité plus haute. Ce label de sécurité est une application restreinte du modèle de BLP ;
- *No Write Up*, s'applique ici sur les sujets du système : les sujets ayant une intégrité strictement inférieure à celle des sujets ne peuvent pas les modifier, c'est l'application d'une des deux propriétés énoncées par Biba ;
- *No Execute Up* : restreint la permission d'exécution sur les objets par les sujets ayant un niveau d'intégrité bas.

La figure 2.13 illustre l'application du modèle de protection MIC.

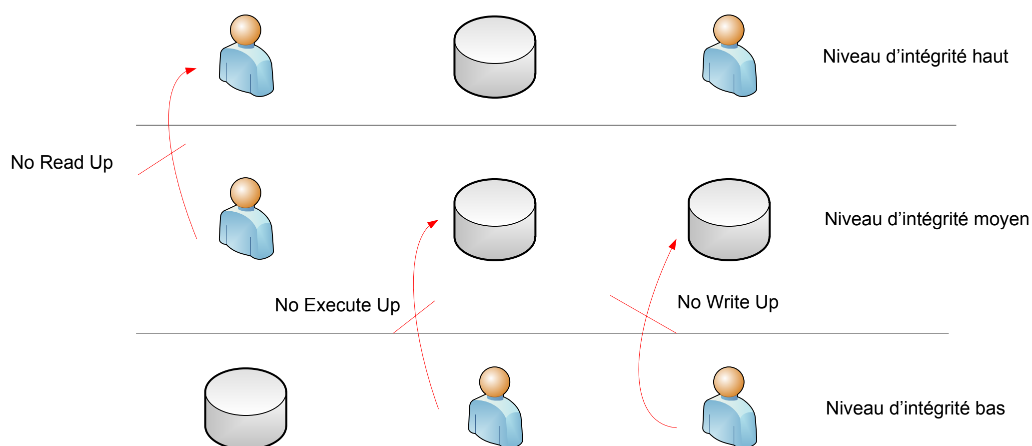

FIGURE 2.13 – Modèle de protection des données : MIC

Cependant, dans le but d'assurer le fonctionnement du système, il existe des exceptions possibles. Les utilisateurs autorisés à effectuer des tâches d'administration ont la possibilité de modifier leur niveau d'intégrité. L'objectif étant d'augmenter le niveau d'intégrité d'un processus pour qu'ils puissent réaliser les tâches privilégiées. Cette augmentation de privilège s'opère, par exemple, lorsqu'un utilisateur veut installer un nouveau logiciel ou réaliser une mise à jour.

Cette autorisation d'augmentation du niveau d'intégrité est représentée par le jeton d'accès **Administrateur**. Il autorise ainsi les processus à écrire dans les répertoires modifiables uniquement par l'administrateur. Depuis Windows Vista, il n'existe plus de compte **Administrateur** sous Windows. On peut rapprocher la possibilité d'augmentation du niveau d'intégrité pour un utilisateur de la commande `sudo` sous Linux, qui permet de changer l'environnement de l'utilisateur, généralement dans le but d'effectuer des tâches d'administration.

Ce modèle de contrôle d'accès est mis en place pour protéger le système et les fichiers systèmes des attaques qui pourraient modifier leur intégrité. À la différence des implantations de mécanisme de contrôle d'accès sous Linux utilisant les LSM, le contrôle du MIC est fait avant le contrôle des droits d'accès discrétionnaire. Cela entraîne la possibilité pour l'utilisateur élevant son niveau d'intégrité par le jeton d'accès Administrateur de contourner les droits d'accès classiques. En effet, une fois son intégrité modifiée, il se retrouve avec des privilèges administrateur ce qui lui permet de contourner les droits d'accès discrétionnaire et lui offre presque tous les droits.

Cette implantation d'un mécanisme de contrôle d'accès obligatoire est la première réalisée par Microsoft. Elle a l'avantage de proposer une approche plus simple que SELinux ou grsecurity puisqu'il n'est pas nécessaire d'écrire une politique énumérant tous les privilèges. En pratique, il

ne protège pas les données des utilisateurs accessibles via le jeton d'accès Administrateur. Ainsi, MIC ne permet pas de minimiser les privilèges, ce qui autorise de nombreuses violations de confidentialité et d'intégrité car il n'implante qu'une partie du modèle de protection de Biba.

2.3.2.2 PRECIP

PRECIP (*Practical and Retrofittable Confidential Information Protection*) [Wang *et al.*, 2008] est un mécanisme de gestion des flux d'information directs développé pour Windows XP. Il implante une simplification du modèle de BLP. En plus d'un mécanisme de contrôle, il est fourni avec une suite d'outils aidant à son administration.

Modèle général

PRECIP associe deux niveaux de sécurité aux objets du système. Ces niveaux sont "haut" et "bas". Les sujets possèdent aussi deux niveaux de sécurité *trust* et *untrusted*. Pour qu'un sujet puisse accéder à un objet ayant un niveau haut, il doit être considéré comme "de confiance".

Cette modélisation est une implantation sur un système réel, ici testée sur Windows XP, du modèle de BLP. L'objectif de cette implantation est de respectée la confidentialité du système, même en cas d'attaque.

Cette implantation d'un mécanisme de contrôle d'accès obligatoire est l'une des premières présente sur les systèmes d'exploitation Windows. Elle implante une simplification de modèle de confidentialité définie par BLP. Cette implantation se base sur le suivi des flux implicites et explicites et sur l'utilisation de l'étiquette dynamique des données. Cependant, le modèle est limité à seulement deux niveaux de confidentialité. De plus, les règles sont limitées et ne protègent pas le système complet, juste les fichiers considérés comme sensibles. Les règles ne sont pas applicables à tout le système et elle ne sont pas extensibles. Enfin, les résultats montrent un impact non négligeable sur les performances du système d'exploitation ainsi que sur le fonctionnement des applications [Wang *et al.*, 2008].

Politique et règle de contrôle d'accès

PRECIP définit deux niveaux de confidentialité pour les objets du système : *high* et *low*. Ces informations sont stockées dans les flux NTFS des objets. Les sujets peuvent avoir deux niveaux de sécurité : *trust* et *untrusted*. Ces niveaux sont stockés dans une structure du noyau. La détermination de ces niveaux est faite sur l'établissement d'une base de données fournie par l'outil.

Un sujet est considéré comme *de confiance* si l'outil a pleinement conscience de toutes les entrées et toutes les sorties possibles de l'exécutable. Si ce n'est pas le cas, il sera alors marqué comme *de non confiance*. Les niveaux de confidentialité des objets du système sont dérivés des droits d'accès discrétionnaire. Par exemple, un fichier dont le propriétaire est l'administrateur est considéré comme sensible.

La définition des niveaux de confidentialité peut aussi être faite par le propriétaire des fichiers grâce aux outils fournis par PRECIP. Des outils spécifiques, permettant de faire transiter des processus en mode sensible comme les navigateurs Internet qui vont sur les sites sensibles, ont aussi été mis en place.

2.3.2.3 Core Force

Core Force [Labs, 2005] est une suite d'outils divisée en deux parties : un pare-feu et un mécanisme de contrôle d'accès. Core Force permet de confiner les processus et contrôler leurs privilèges.

```
1 <Programs>
2   <Program Name="Mozilla Firefox 1.0" SecurityLevel="MediumLow" Status="
      Enabled">
```

Listing 2.11 – Extrait d'un fichier de configuration de Core Force pour un programme

```
1     <Policies>
2       <Policy Name="General. Environment for Mozilla Firefox">
3         <Documentation>A Policy includes previously defined configuration
              parameters.</Documentation>
4         <Permissions>
5           <FileSystem>
6             <Permission Read="Yes" Write="Yes">
7               <PredefinedFolder Dir="Mozilla" File="pluginreg.dat" Name="
                  ApplicationSpecificData"/>
8             </Permission>
```

Listing 2.12 – Extrait d'un fichier de configuration de Core Force pour un programme spécifiant les permissions

Modèle général

Core Force se base sur le modèle de grsecurity pour faire du contrôle d'accès, c'est-à-dire qu'il va appliquer le triplet (r,w,x) pour chaque objet du système. Comme grsecurity ou SELinux, il est nécessaire de définir explicitement les interactions autorisées pour chaque application du système.

Core Force est fourni sous la forme d'un logiciel, installable uniquement Windows 2000 et sur Windows XP. Il est composé de deux *driver* noyau, un qui contrôle des interactions sur le système de fichiers et sur le registre, et le second pour contrôler les interactions du réseau.

La configuration par défaut définie des profils de sécurité pour les applications. Ces profils sont des *templates* de sécurité, qui proposent des politiques de sécurité par défaut vis-à-vis du profil sélectionné. Il est cependant possible de créer des profils spécifiques pour les applications.

Politique et règle d'accès

La politique de contrôle d'accès s'appuie sur le chemin complet des sujets et des objets. Tout comme grsecurity, une règle de contrôle d'accès associe un sujet à un objet avec un ensemble de permission qui se définissent grâce aux ACL.

Pour configurer les permissions pour les applications, Core Force propose une interface graphique facilitant la configuration du contrôle d'accès. Les permissions sont ensuite stockées dans des fichiers au format XML.

Nous allons détailler un exemple de politique. Tout d'abord, le fichier XML spécifie pour quel programme s'applique cette politique 2.11.

Ensuite, le fichier de configuration spécifie les interactions autorisées par le programme 2.12. Nous pouvons noter ici que le programme `Firefox` a le droit de lecture et d'écriture sur le fichier `pluginreg.dat` dans le dossier `Mozilla`.

L'approche Core Force a été abandonnée depuis l'arrivée de Windows Vista, 7 et 8.

2.3.3 Distributed Security Infrastructure

Distributed Security Infrastructure (DSI) [Pourzandi *et al.*, 2002] est un *framework* de déploiement et d'administration d'une politique de contrôle d'accès pour un environnement HPC. C'est la première implantation d'un *framework* de gestion d'une politique de sécurité orienté

système distribué. Le projet a ensuite évolué pour être intégré dans un *framework* nommé HA-OSCAR [Leangsuksun et Haddad, 2004].

Les auteurs de DSI ont mis en place une architecture distribuée pour répondre à la problématique de déploiement et de gestion d'une politique de sécurité dans un environnement distribué. L'intégration de DSI dans HA-OSCAR avait pour objectif de cibler les environnements HPC. Cette intégration a entraîné le changement de nom pour devenir RASS [Darivemula *et al.*, 2006] (*Reliability, Availability, Serviceability and Security*). Cette architecture se base sur la définition de deux éléments clés : le *Security Server* et *Security Manager*.

Le *Security Server* est le point d'entrée du système pour la gestion de l'infrastructure. Il est chargé du déploiement de la politique de sécurité sur tous les nœuds. Il doit aussi distribuer les mises à jour de la politique. Il s'occupe enfin de remonter les alertes de sécurité en provenance des nœuds.

Les *Security Manager* sont déployés sur les nœuds du cluster. Ce sont les agents locaux chargés d'appliquer la politique de sécurité qu'ils ont reçue du serveur de sécurité. Dans [Pourzandi *et al.*, 2002], les auteurs appliquent une politique de sécurité pour SELinux.

Les communications entre le serveur de sécurité et les managers se font sur des canaux authentifiés et chiffrés. Ces canaux de communication sont utilisés pour remonter les alarmes et les alertes de sécurité ainsi que pour la distribution de la politique de sécurité.

La politique de sécurité, appelée *Distributed Security Policy* (DSP) par les auteurs, définit explicitement les accès autorisés sur chaque système. Elle est gérée sur le SS par un administrateur. Grâce aux canaux de communication chiffrés, les agents locaux autorisent des mises à jour de la politique. Les règles d'accès définissent les accès autorisés sur le système mais aussi sur les échanges entre les systèmes. Pour contrôler les interactions entre les systèmes, DSI ajoute un *security node ID* (SNID). En plus de ce SNID, DSI définit des *secure ID*(SID) pour chaque sujet (SSID) et chaque objet d'un système (TSID). Grâce à ces deux ID, les auteurs établissent le *cluster security ID* (CSID). Ce CSID est ajouté aux options des paquets IP. Ce sont enfin les *Security Manager* qui réalisent le contrôle d'accès en vérifiant les CSID présents dans chaque paquet IP. Il est ainsi possible de contrôler les communications établies entre les nœuds du cluster. Cependant, cette approche est très proche de SELinux et peu extensible, elle n'offre en pratique aucun couplage possible avec d'autres mécanismes de sécurité.

2.3.4 Impact sur les performances

Nous allons traiter dans cette partie de l'impact sur les performances des mécanismes de contrôle d'accès. En vue de l'intégration d'un mécanisme de contrôle d'accès obligatoire, il est nécessaire de connaître son impact sur le système d'exploitation, spécialement pour le HPC.

Dans l'article [M. Fox et Thomas, 2003], les auteurs opposent SELinux à grsecurity. Ils présentent dans une première partie le fonctionnement de chaque mécanisme de contrôle d'accès obligatoire. Puis ils exécutent différents *benchmarks* : `lmbench` et `unixbench`. Ces *benchmarks* réalisent des séries d'opérations sur le système de fichiers comme des lectures et écritures de fichiers de différentes tailles mais aussi des opérations spécifiques comme des exécutions de processus, etc. Ces *benchmarks* sont exécutés suivant différentes configurations (par un utilisateur standard, un administrateur, puis en mode protection). La conclusion de cet article est que ces mécanismes de contrôle d'accès influencent différemment les performances du système d'exploitation. Cette influence va dépendre de l'action réalisée : écriture, changement de contexte, etc. Il n'est donc pas possible de dire avec précision si un mécanisme de contrôle d'accès est plus adapté qu'un autre aux environnements HPC. La principale limite de cet article est que les auteurs ne font que comparer SELinux avec grsecurity sans proposer les résultats pour un système sans contrôle d'accès obligatoire.

Les auteurs de RASS [Leangsuksun *et al.*, 2005] proposent aussi une étude sur l'impact sur les performances du mécanisme de contrôle d'accès obligatoire qu'ils ont mis en place dans un milieu HPC. Les résultats portent sur des versions du noyau Linux assez anciennes puisque la version du noyau est 2.4.17.

L'impact sur le système de fichiers est compris entre 0% et 2% suivant les opérations par rapport à un système sans le système *RASS*. Les opérations les plus impactées sont les exécutions de nouveau processus. Par contre, une latence comprise 20% et 30% a été mesurée pour les messages envoyés et reçus en UDP. Cette latence est due aux modifications des paquets IP faites par les *Security Manager* dans le but d'y ajouter les CSID. Sans le contrôle réseau, la latence est de l'ordre de 5%.

Cette première étude montre que l'intégration d'un mécanisme de contrôle d'accès obligatoire dans un environnement ayant de fortes contraintes en termes de performances est possible. Cependant, les latences réseaux générées à cause des modifications des paquets IP ne sont pas encore intégrables dans les environnements HPC.

Dans une seconde étude [Blanc et Lalande, 2012], les auteurs montrent que l'intégration d'un MAC peut être faite dans un environnement HPC. Dans une première partie, les auteurs proposent la mise en place de propriétés de sécurité pour confiner les utilisateurs et protéger les services critiques présents sur le système. Pour cela, ils utilisent SELinux. Le confinement des utilisateurs se fait par l'utilisation des catégories présentes dans SELinux. Ils concluent par des tests de performance sur le système de fichiers *Lustre*. Il en résulte que l'ajout de SELinux, associé à ce système de fichiers spécifiques, engendre un surcoût de 10% sur les entrées/sorties. Cependant, il s'agit uniquement d'une expérimentation avec SELinux qui n'offre pas de cadre général pour évaluer les performances.

2.3.5 Discussion

Nous venons de détailler les implantations des mécanismes de contrôle d'accès obligatoire présents sur les systèmes d'exploitation Linux et Windows. Pour chaque implantation, nous avons détaillé les propriétés de sécurité que le mécanisme est capable de garantir.

Le tableau 2.2 récapitule, pour chaque système d'exploitation étudié, la liste des propriétés de sécurité supportées. Nous pouvons tirer deux constats de ce tableau. Le premier est que les mécanismes de contrôle d'accès obligatoire présents sous Linux sont assez complets et permettent de couvrir un large panel de propriétés de sécurité que nous avions exprimés. Il n'est donc pas nécessaire de rajouter un nouveau mécanisme pour ce système d'exploitation. Cependant, nous avons aussi montré un manque d'intégration de ces mécanismes dans les environnements distribués et plus particulièrement dans les environnements HPC. Seul le projet DSI propose une intégration d'un MAC dans les clusters, mais sans pour autant juger avec précision de l'impact sur les performances du système. De plus, les travaux proposés sont maintenant assez anciens par rapport aux noyaux Linux actuels.

Le second constat que nous pouvons faire, est le manque de mécanismes capables d'appliquer des propriétés de sécurité sur les systèmes Windows. Le plus complet est le MIC, le mécanisme mis en place par Microsoft, mais ce dernier souffre de quelques faiblesses qui peuvent se révéler dangereuses pour le système. Comme nous l'avons détaillé, ce mécanisme n'est pas capable de protéger efficacement tout le système puisqu'il s'attache à vérifier l'intégrité des fichiers du système. Il ne contrôle pas, par défaut, les actions effectuées sur les fichiers des utilisateurs. Certains mécanismes tels que Core Force ou PRECIP existent, mais ils ne sont, soit plus maintenus pour les systèmes actuels (Windows 7 et Windows 8) soit ne permettent pas un contrôle précis de toutes les activités.

	Linux				Windows		
MAC	SELinux	grsecurity	Tomoyo	PIGA	MIC	PRECIP	Core Force
Confidentialité							
- des objets	oui	oui	oui	oui	x	oui	x
- des sujets	x	x	oui	oui	oui	x	x
Intégrité							
- des objets	oui	oui	oui	oui	oui	x	x
- des sujets	x	x	x	oui	oui	x	x
Propriétés spécifiques							
- BLP	oui	oui	x	oui	x	oui	x
- Biba	oui	x	x	oui	oui	x	x
Propriétés dérivées							
- Conf de processus	oui	oui	oui	oui	x	oui	oui
- Moindre privilège	oui	oui	oui	oui	x	x	oui
- Séparation de privilèges	oui	oui	x	oui	x	x	x
- Non-interférence	x	x	x	oui	x	x	x
- Nouvelles propriétés	oui	oui	oui	oui	x	x	oui

TABLE 2.2 – Tableau récapitulatif des propriétés de sécurité appliquées par les implantations de contrôle d'accès obligatoire

2.4 Conclusion

La problématique du contrôle d'accès s'avère différente à traiter suivant les systèmes d'exploitation. D'un côté, nous avons plusieurs implantations matures et éprouvées, de l'autre, il n'existe que très peu de choses.

Sur les systèmes d'exploitation Linux, nous avons choisi de détailler l'implantation de quatre mécanismes de contrôle d'accès obligatoire. Il en existe cependant d'autres proposant leurs propres implantations. Nous pouvons par exemple citer SMACK [Casey Schaufler , 2008] (*Simplified Mandatory Access Control Kernel*) qui se base sur la définition de labels de sécurité et qui propose une implantation du modèle de BLP. Comme son nom l'indique, c'est un mécanisme qui se veut simple à configurer et à utiliser puisqu'il ne propose que peu de labels (trois par défaut). Il est notamment utilisé dans le système d'exploitation de Samsung pour ses téléphones portables, appelé Tizen.

Tous ces mécanismes se basent sur la définition de politique de sécurité qui peuvent être rédigées soit à la main, soit générées de manière plus ou moins automatiques facilitant ainsi grandement leur intégration dans les systèmes Linux. La mise à disposition d'outils d'administration et d'aide à la gestion des politiques permet d'avoir des systèmes qui peuvent être maintenus facilement. De plus, il est facile de noter que sur les systèmes d'exploitation Linux, nous avons le choix dans le mécanisme de contrôle d'accès à utiliser. Ce choix est en partie dû à la possibilité de s'intégrer au noyau facilement en utilisant les LSM, hormis grsecurity qui continue de proposer un patch noyau.

Sur les systèmes d'exploitation Windows, il n'existe pas autant de choix. Les premiers projets sur le renforcement du contrôle d'accès ne proposent que d'étudier les faiblesses de configuration du contrôle d'accès discrétionnaire, par exemple Netra [Naldurg *et al.*, 2006]. Ces outils, qui fournissent des rapports et des graphiques présentant les attaques possibles à partir d'un état du système, ne sont que des moyens de détection hors ligne et n'offrent pas de réelles solutions pour renforcer la sécurité des systèmes d'exploitation Windows.

D'autres projets proposent des mécanismes pour faire de la protection du système. Ces implantations se focalisent sur certains points précis du système : gestion des exécutables ou protection des fichiers considérés comme sensibles. Ces outils sont issus de phase d'expérimentation sur des

logiciels malveillants, ils ne sont pas capables de contrer les comportements qu'ils ne connaissent pas.

Le mécanisme de contrôle d'accès obligatoire proposé par Microsoft, le MIC, vise essentiellement à protéger le système de toute modification malveillante, mais ce mécanisme ne protège en aucun cas les données des utilisateurs (protection en intégrité ou en confidentialité). De plus, dans la pratique, l'utilisation de ce mécanisme se révèle difficile pour l'ensemble du système et fragile puisque l'utilisateur peut autoriser l'élévation de privilèges. Nous pouvons aussi ajouter que Microsoft propose, sur les versions *Enterprise* et *Ultimate* de Windows 7, un outil qui se nomme AppLocker [Microsoft, 2009]. Il permet de configurer les droits d'accès sur les ressources du système. Cette configuration peut se faire de manière graphique.

Nous nous sommes aussi intéressés à l'intégration dans les environnements distribués, et plus particulièrement dans les milieux HPC, des mécanismes de contrôle d'accès obligatoire. Il existe des projets visant à renforcer la sécurité des grilles, notamment les échanges entre les différents services par exemple en utilisant les langages de type XACML [OASIS, 2013]. Néanmoins, si le projet DSI s'intéresse à l'intégration d'un MAC dans les milieux HPC, il traite de façon incomplète l'impact d'un tel mécanisme sur les performances du système d'exploitation.

D'une part, nous avons donc vu que sur les systèmes d'exploitation Linux, il existait un certain nombre de modèles de protection obligatoire. Tous ces modèles ont été développés pour être intégrés dans les systèmes pour les stations de travail. Avec le développement des systèmes distribués, mais aussi des environnements de calcul à haute performance, le renforcement du contrôle d'accès sur les grilles devient une nécessité. Peu de projets se sont intéressés à l'intégration de ce type de mécanisme de contrôle d'accès dans les environnements distribués et l'impact d'un tel mécanisme sur les performances des systèmes d'exploitation est peu évoqué. On manque notamment de modèles et de protocoles précis d'évaluation des performances. Enfin, nous considérons qu'il n'est pas envisageable de faire reposer le contrôle d'accès sur un seul mécanisme et là encore, les approches et les modèles permettant d'associer différents mécanismes de contrôle d'accès manquent. De plus, les modèles de répartition de ces mécanismes sont quasiment inexistants. Enfin, si déjà les protocoles d'évaluation des performances manquent pour les systèmes de protection homogènes (par exemple basés exclusivement sur SELinux), ils sont encore plus inexistants dans le cadre des systèmes répartis de contrôle d'accès hétérogènes où l'on voudra associer et répartir différents mécanismes.

D'autre part, nous avons noté le manque de mécanisme de contrôle d'accès obligatoire sur les systèmes Windows capable d'appliquer des propriétés de sécurité que nous avons définies au début de ce chapitre. La nécessité de l'implantation d'un tel mécanisme dans les systèmes Windows est extrêmement grande puisque c'est, encore à ce jour, le système le plus utilisé en tant que station de travail.

Les chapitres suivant traitent de façon uniforme de ces différents points en s'intéressant à la fois à la protection des systèmes d'exploitation hétérogènes, à l'association des différents mécanismes de protection, à leurs répartitions et aux méthodes et protocoles d'évaluation des performances dans ces systèmes répartis de protection.

Chapitre 3

Modélisation et répartition d'observateurs

Les exemples que nous avons détaillés dans l'introduction de notre étude (voir la section 1.3), ont mis en avant que les attaques sur les systèmes d'exploitation sont de plus en plus complexes. Pour les prévenir de manière efficace, c'est-à-dire sans nuire au fonctionnement du système, il faut connaître le **scénario complet** de ces attaques.

La connaissance de ce scénario complet passe par le contrôle des interactions directes au sein du système. Pour contrôler ces interactions, la mise en place d'un **observateur**, capable de détourner le flux d'exécution régulier et de prendre des décisions en fonction d'une politique de sécurité, est essentielle. Ce modèle de protection, basé sur deux éléments complémentaires, s'appuie pleinement sur la définition d'un *Policy Enforcement Point* et d'un *Policy Decision Point*. Le PEP est le moyen utilisé par l'observateur pour détourner le flux d'exécution. On peut citer par exemple les LSM, présents nativement sur les systèmes Linux, qui offrent la possibilité au module de sécurité de détourner les flux d'exécution réguliers. Le PDP est la partie de l'observateur qui prend les décisions. Dans le cadre d'un mécanisme contrôlant les accès, [Anderson, 1980] a défini un observateur particulier : le **moniteur de référence**. Le moniteur de référence est donc un observateur prenant des décisions d'accès au regard d'une politique de sécurité.

Mais le seul contrôle des interactions directes ne suffit pas à protéger le système contre les scénarios complets. En effet, un observateur ne traitant que les interactions directes n'est pas capable de traiter un scénario complet d'attaque. Il faut pour cela utiliser d'autres observateurs capables de reconstruire le scénario complet. Pour ce faire, un premier type d'observateur doit être capable **d'horodater** chaque interaction directe dans le but de les situer les unes par rapport aux autres. Ensuite, il faut être capable d'**associer** les différents observateurs nécessaires à la reconstruction du scénario pour qu'ils puissent s'échanger non seulement les requêtes, mais aussi les décisions d'accès qu'ils prennent.

Pour prendre sa décision, un observateur s'appuie sur une politique de sécurité. Cette politique peut être une base de données contenant les éléments à bloquer, des signatures à détecter, ainsi que les interactions directes à contrôler. Comme nous avons besoin en premier lieu d'un observateur capable de gérer les interactions directes, nous allons proposer un modèle de politique de sécurité générique pour contrôler les accès directs des processus sur les ressources. Nous appliquerons plus spécifiquement cette politique pour les systèmes Windows car, comme nous l'avons montré au cours de l'état de l'art 2.4, c'est sur ce système qu'il y a le plus de manque. Dans le but d'obtenir une politique de sécurité générique pour les systèmes Windows, nous proposons une désignation des ressources pour ces systèmes.

Le langage pour la politique directe supporte à la fois PBAC et DTE. Dans le modèle PBAC, les ressources sont identifiées par leur chemin complet dans l'arborescence du système de fichiers,

comme le fait grsecurity. Pour PBAC, notre désignation des ressources permet une politique portable car indépendante de la localisation des ressources. Dans le modèle DTE, les ressources sont identifiées par des types et des domaines. Cette méthode supporte mieux l'hétérogénéité puisque la politique pourrait aisément être projetée sur Linux comme sur Windows.

Nous étudierons ensuite comment mettre en œuvre un PEP pour Windows. Ainsi grâce à ce PEP et à la politique générique, nous pouvons proposer un PDP qui gère les interactions directes et offre ainsi un moniteur de référence pour les versions récentes de Windows, à savoir Windows 7.

3.1 Définition d'un observateur pour les systèmes d'exploitation Linux et Windows

Cette section s'intéresse à la définition générique de la notion d'observateur. Pour que l'observateur puisse fonctionner sur le système, cela passe par la mise en place de deux éléments :

— un PEP (*Policy Enforcement Point*, point d'application de la politique) : c'est-à-dire un moyen pour détourner le flux d'exécution régulier du système ainsi que pour appliquer les décisions prises par l'observateur ;

— une politique de sécurité utilisée par le PDP (*Policy Decision Point*, un point de décision), qui définit les traitements à effectuer.

Dans le cadre d'un observateur contrôlant les accès au sein d'un système, l'application de la politique de sécurité se fait par un observateur particulier nommé **moniteur de référence**. Mais pour que ce contrôle soit efficace, il est nécessaire que le moniteur puisse observer l'ensemble du système. Pour cela, l'observateur se place de manière naturelle en espace noyau.

L'espace noyau est une protection matérielle qui a été mise en place au niveau des processeurs. Cette protection se nomme **anneau de protection** (ou *rings*). L'objectif de cette protection est d'établir des niveaux de confiance pour l'exécution de code. Les systèmes d'exploitation modernes n'utilisent que deux anneaux de protection : l'anneau 0 et l'anneau 3. L'anneau 0 est un anneau de protection considéré *de confiance*, c'est-à-dire que le code qui s'exécute n'est pas supervisé. Cela se traduit par la possibilité de réaliser des tâches privilégiées et de pouvoir utiliser toutes les instructions du processeur. C'est dans cet anneau de protection que s'exécute le noyau du système d'exploitation. L'anneau de protection 3 est un anneau de protection où s'exécute du code qui n'est pas considéré *comme sûr*. Cela se traduit par la possibilité d'utiliser un jeu d'instructions plus restreintes au niveau du processeur. Les systèmes d'exploitation modernes utilisent l'anneau de protection 3 pour exécuter les applications. Dans ce mode, les interactions des applications sont supervisées et, par conséquent, pour réaliser une opération spécifique, l'application doit passer par un **appel système**, qui lui sera contrôlé et exécuté par le noyau.

C'est dans le but de pouvoir réaliser des tâches privilégiées que nous plaçons l'observateur en espace noyau. Il pourra ainsi contrôler tout le système sans restriction. Les opérations d'entrée/-sortie n'étant possibles que par les appels système, le détournement des appels systèmes permet de contrôler toutes les interactions entre des processus sur les ressources.

Sur les systèmes d'exploitation Windows, pour exécuter du code en espace noyau, il faut que ce code soit directement dans le noyau ou soit exécuté par un pilote de périphérique (un *driver*). Dans le cas de Windows, comme il n'est pas possible de recompiler le noyau pour y inclure nos modifications, nous sommes contraints de mettre en place un *driver*. Sur les systèmes d'exploitation Linux, ce contrôle peut être effectué soit par un module noyau, soit par une modification du code du noyau, puisque ce dernier est modifiable.

La figure 3.1 illustre le placement de l'observateur de manière générale vis-à-vis d'un appel système fait par un processus. Lorsqu'un processus effectue une interaction sur une ressource, celle-ci passe par un appel système (flèche 0). Dans un premier temps, l'observateur doit détourner

le flux d'exécution (flèche 1). Il prend ensuite une décision au regard de sa politique de sécurité (flèche 2). Dans le cas où l'interaction est autorisée, il va exécuter l'appel système et récupérer le retour de celui-ci (flèche 3). L'observateur peut aussi réaliser des traitements spécifiques sur le retour de l'appel. Pour finir, et quelle que soit le traitement que l'observateur a réalisé, il renvoie une réponse au processus ayant réalisé l'interaction (flèche 4).

Un observateur est donc composé d'un mécanisme de détournement appelé *Policy Enforcement Point* et d'un mécanisme de prise de décision nommé *Policy Decision Point*.

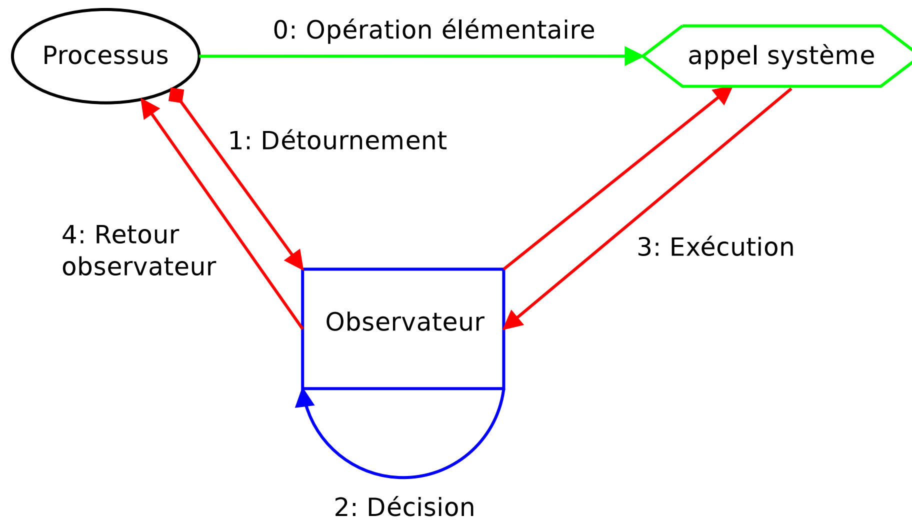

FIGURE 3.1 – Schéma du placement de l'observateur

3.1.1 Éléments de base de la protection système

Pour définir de façon générique la notion d'observateur, nous allons nous appuyer sur les deux scénarios d'attaque que nous avons présentés en introduction de ce mémoire (voir les figures 1.2 et 1.3). De plus, nous allons préciser les éléments de base de la protection système, non seulement pour un observateur mais aussi pour l'écriture de la politique de sécurité.

Dans ces deux scénarios, nous observons une interaction directe commune : l'écriture par un navigateur web d'un fichier malveillant comme décrit par la figure 3.2.

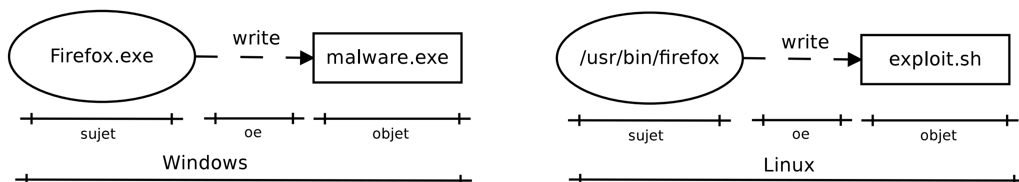

FIGURE 3.2 – Écriture d'un fichier malveillant par `Firefox` : Windows et Linux

Cette interaction est composée de trois éléments. Le premier élément est le **sujet**, correspondant au processus `Firefox`, le second élément est l'**objet**, le fichier `malware.exe` sur Windows et `exploit.sh` sur Linux, et une **opération élémentaire** `write`. Cette interaction directe est représentée de la manière suivante :

$$Firefox \xrightarrow[write]{} malware.exe$$

Définition 3.1.1 ***Un sujet*** *Un sujet* $\mathbf{s}$ *est une entité active du système qui réalise une opération élémentaire. Sur les systèmes Windows, l'ensemble des sujets* $\mathbf{S}$ *est composé des processus, des services, des services système et des* drivers. *Sur les systèmes Linux, ce sont exclusivement les processus.*

En nous appuyant sur la figure 3.2, le processus nommé `Firefox.exe` est le sujet pour les systèmes Windows alors qu'il est nommé `/usr/bin/firefox` sur Linux. Sur les systèmes Windows, les processus sont représentés par leur nom relatif et non par leur chemin complet.

Définition 3.1.2 ***Un objet*** *Un objet* $\mathbf{o}$ *est une entité du système qui est la cible d'une opération élémentaire. L'ensemble des objets* $\mathbf{O}$ *est composé des ressources classiques (fichiers, répertoires,* socket*, etc.), des éléments du registre (les ruches, les clés, les valeurs et des données) ainsi que de l'ensemble* $\mathbf{S}$ *des sujets.*

Dans la figure 3.2, le fichier `malware.exe` est un exemple d'objet pour Windows et le fichier `exploit.sh` l'est pour Linux.

Définition 3.1.3 ***Une classe*** *Une classe* $\mathbf{class}$ *est utilisée pour caractériser un objet. Elle permet de spécifier le type d'objet, et par conséquent, les opérations élémentaires possibles sur cet objet. L'ensemble* $\mathbf{Class}$ *constitue l'ensemble des classes autorisées sur le système.*

La classe permet ainsi de différencier les opérations élémentaires réalisées sur un fichier des opérations réalisées sur un tube. Les objets `malware.exe` et `exploit.sh` sont de la classe **fichier**.

Définition 3.1.4 ***Une opération élémentaire*** *Une opération élémentaire* $\mathbf{oe}$ *est une opération de base effectuée par un sujet sur un objet (ou sur un sujet) lors de l'exécution d'un appel système.*

Par exemple sur Linux, l'**ouverture** d'un fichier met en œuvre deux opérations élémentaires. L'opération élémentaire **getattr** vérifie les droits sur le fichier et l'opération élémentaire **open** retourne un descripteur de fichier.

Définition 3.1.5 ***Un appel système*** *Un appel système est une fonction du noyau permettant d'effectuer des opérations d'entrée/sortie, de communication, etc. Toutes ces opérations sont considérées comme sensibles au niveau du système d'exploitation. C'est pour cette raison qu'elles sont réalisées par le noyau. Lors d'un appel système, une opération élémentaire est réalisée.*

Pour pouvoir réaliser l'opération élémentaire **write**, le processus `Firefox.exe` utilise une fonction nommée `WriteFile()`, présente en espace utilisateur. Elle correspond à un appel système en espace noyau qui se nomme `NtWriteFile()`. C'est cette fonction qui réalise l'opération d'entrée/sortie sur le fichier. Nous obtenons la même chose sous Linux de manière complètement analogue, seuls les noms des fonctions sont différents. La figure 3.3 détaille ce cheminement d'une écriture dans un fichier.

La définition des notions de sujets, d'objets et d'opérations élémentaires, nous permettent de formaliser la notion d'**interaction**.

Définition 3.1.6 ***Une interaction*** *Une interaction* $\mathbf{it}$, $s \underset{oe}{\longrightarrow} o$ *est un triplet composé d'un sujet, d'un objet et d'une opération élémentaire. Elle est notée* $\mathbf{it} = (\mathbf{s}, \mathbf{o}, \mathbf{oe})$

Dans notre exemple 3.2, l'interaction est composée du sujet `Firefox.exe`, de l'opération élémentaire `écriture` et de l'objet `malware.exe`.

Nous pouvons affiner cette définition d'**interaction** par l'ajout de la notion **directe**. Cela signifie que le sujet exécute directement un appel système sur l'objet.

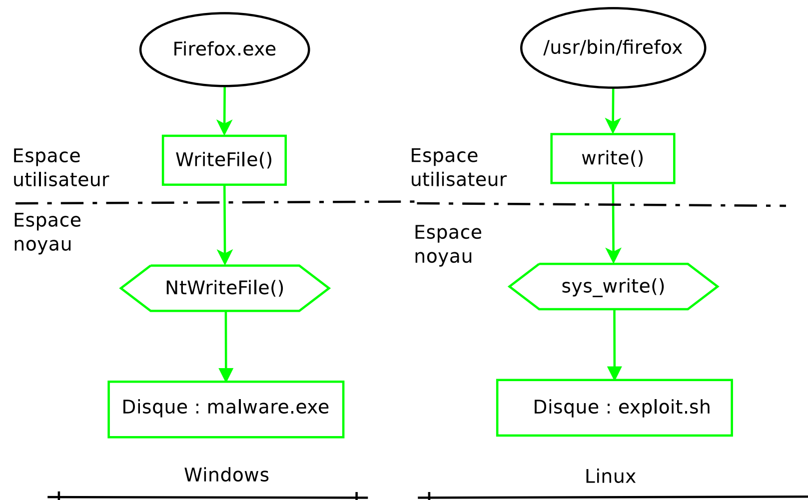

FIGURE 3.3 – Écriture détaillée d'un fichier malveillant par un navigateur web

Définition 3.1.7 ***L'observateur*** *Pour observer une interaction directe, il est nécessaire de mettre en place un mécanisme capable d'intercepter les opérations élémentaires. Pour cela, cette interception est réalisée par un observateur au niveau des appels système. Cet observateur doit détourner le flux d'exécution. La figure 3.4 montre le placement de l'observateur en amont de l'appel système, donc en espace noyau. En se plaçant en amont de l'appel système, l'observateur est capable de bloquer le flux d'exécution dans le cadre de l'application d'une politique de protection. Il peut aussi traiter le retour de l'appel système.*

Il existe différents types d'observateurs : les moniteurs de référence, qui contrôlent les interactions, les auditeurs, qui génèrent des traces en fonction des interactions, les vérificateurs d'intégrité, etc.

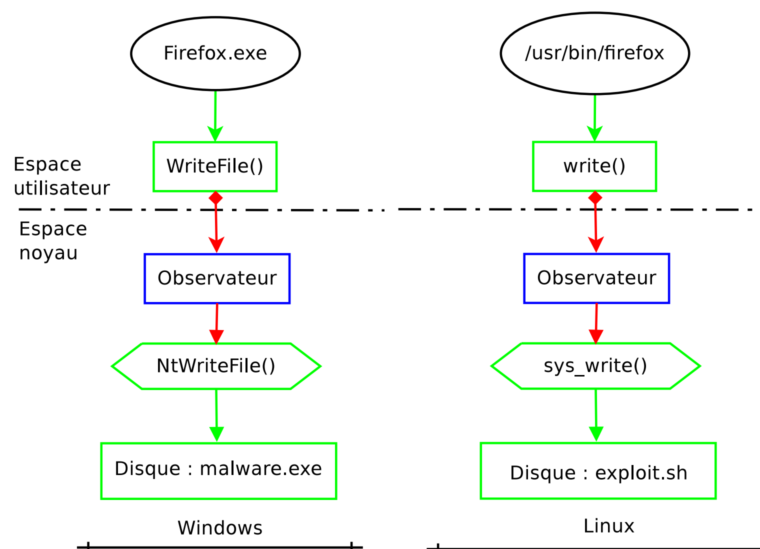

FIGURE 3.4 – Placement de l'observateur pour détourner le flux d'exécution

Nous proposons le placement d'un observateur en espace noyau car nous voulons avoir un observateur capable de contrôler toutes les interactions qui se passent sur le système, tout en se protégeant des potentielles attaques des processus.

Définition 3.1.8 ***Détournement de flux d'exécution*** *Le détournement du flux d'exécution est une opération réalisée le plus souvent en espace noyau dans le but de contrôler les interactions réalisées par le système. Pour réaliser cela, il faut, soit utiliser un mécanisme prévu dans le système d'exploitation comme les* LSM*, soit modifier directement le noyau.*

Lors de sa prise de décision, l'observateur a la possibilité d'enregistrer sa décision dans le but d'assurer la traçabilité des interactions qu'il observe. Pour cela, il génère une **trace**.

Définition 3.1.9 ***Trace*** *Une trace* $\mathbf{tr}$ *est générée par l'observateur lors d'une observation, elle contient l'interaction* $\mathbf{it}$ *décrivant l'opération. Elle peut aussi contenir d'autres informations renseignant sur l'état du système comme un identifiant de trace ou un horodatage que l'on peut noter* $\mathbf{d}$. $\mathbf{tr} = (\mathbf{d}, \mathbf{s}, \mathbf{o}, \mathbf{oe})$.

Pour réaliser un traitement spécifique vis-à-vis d'une observation, l'observateur s'appuie sur une **politique de sécurité**.

Définition 3.1.10 ***La politique de sécurité*** *Un observateur a besoin d'une politique de sécurité dans le but de réaliser des traitements spécifiques.*

Une politique de sécurité peut être de la forme : politique d'audit, politique de détection, politique de protection, etc. Une politique est constituée d'un ensemble de règles. Une règle permet à l'observateur d'exécuter un traitement spécifique lorsqu'il intercepte une interaction. Il existe plusieurs types de règles : règles d'audit, règles de détection, règles de protection, etc.

Dans le cas d'une politique de protection, le traitement est le suivant : si l'interaction courante ne correspond pas à une règle d'autorisation dans la politique de sécurité, alors l'observateur l'interdira.

La figure 3.5 illustre le fonctionnement du système avec la mise en place d'un observateur ainsi que d'une politique de sécurité. Nous avons vu que les deux systèmes d'exploitation étudiés, Windows et Linux, possèdent une symétrie au niveau de leur fonctionnement. Grâce à cette symétrie, nous pouvons obtenir un placement générique pour notre observateur commun au deux systèmes.

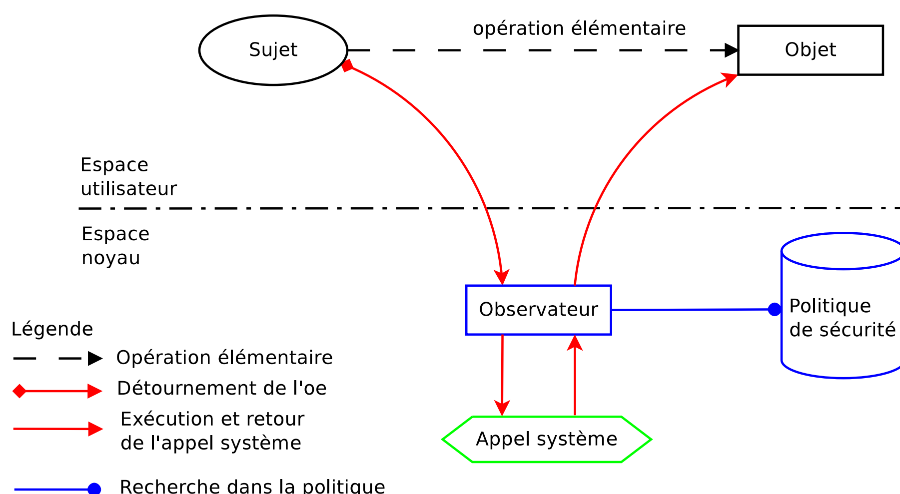

FIGURE 3.5 – Schéma global de détournement des flux pour les deux systèmes

Chaque interaction est détournée par l'observateur en espace noyau qui exécute un traitement conforme à sa politique de sécurité. Dans le cas où l'interaction n'est pas bloquée, l'appel système se poursuit.

3.1.1.1 Modélisation du système observé

Nous venons de définir les éléments clés d'un système : les sujets, les objets, les opérations élémentaires, etc. Grâce à ces définitions, nous allons pouvoir modéliser l'observation du système d'exploitation.

Le système est constitué d'un ensemble S de sujets et d'objets O. Les sujets réalisent des opérations élémentaires sur les objets. Ces opérations se traduisent par la définition d'une interaction $it = (s, o, oe)$. L'ensemble des interactions possibles sur un système est représenté par l'ensemble IT. Nous pouvons noter que lors d'un appel système, une opération élémentaire oe est réalisée entre l'appelant (le sujet) et la cible (l'objet ou le sujet). Ainsi lors d'un appel système, l'observateur reconstruit une interaction directe.

Un observateur détourne un appel système pour prendre une décision et reconstruit une interaction. Cette décision s'appuie sur une politique de sécurité Pol, qui est composée d'un ensemble de couples : $traitement_observateur$ et it formant les règles d'accès. On obtient alors, pour une politique Pol donnée, $Pol = \sum(traitement_observateur, it)$. À partir de l'interaction observée et d'une politique de sécurité, l'observateur prend une décision en utilisant la fonction Dec : $Dec(it, Pol) \mapsto (decision, tr)$ où tr est une trace générée par l'observateur comme le détaille l'algorithme 1.

Algorithm 1 Prise de décision de l'observateur

Require: Pol
Require: it
  $d \leftarrow Horodatage()$
  $decision \leftarrow RechercheDansPolitique(it, Pol)$
  $tr \leftarrow GenererTrace(d, decision, it)$
  return $decision, tr$

L'observateur s'appuie sur deux fonctions : $RechercheDansPolitique$, fonction qui recherche l'interaction observée dans la politique pour en extraire une décision et $GenererTrace$, qui construit une trace.

L'algorithme 2 détaille le mécanisme de recherche d'une interaction it dans une politique Pol de sécurité. Une politique Pol est composée d'un ensemble de couples : interaction it et traitement $traitement_observateur$. Lorsque l'interaction n'existe pas dans la politique, la fonction retourne un traitement par défaut $traitement_default$ qui sera spécifié en fonction du type d'observateur voulu.

Algorithm 2 Recherche dans la politique

Require: Pol
Require: it
  $trouve \leftarrow False$
  for all $r \in Pol$ ET $trouve == False$ **do**
    if $Satisfait(it, r)$ **then**
      $trouve \leftarrow True$
    end if
  end for
  if $trouve == True$ **then**
    $decison \leftarrow r.traitement_observateur$
  else
    $decision \leftarrow traitement_default$
  end if
  return $decision$

Enfin, la deuxième fonction sur laquelle s'appuie l'observateur est la fonction générant une trace tr, détaillée par l'algorithme 3. Elle prend en paramètre l'horodatage, la décision ainsi que l'interaction. Elle ajoute à ces trois éléments un identifiant unique de trace.

Algorithm 3 Génération d'une trace

Require: d (horodatage)
Require: $decision$
Require: it
  $id \leftarrow CurrentId()$
  $tr \leftarrow (d, id, it, decision)$
  return tr

On peut, à partir de tous ces éléments, modéliser le système observé, comme l'ensemble des sujets, des objets, des interactions auxquels on ajoute une politique de sécurité et un observateur :

$$System_{observe} = (S, O, IT, Pol, Observateur)$$

3.1.1.2 Mode de sécurité

Un observateur possède 3 modes de sécurité. Un mode de sécurité est le comportement de l'observateur vis-à-vis d'une requête qu'il reçoit. Par construction, ces 3 modes de sécurité sont mutuellement exclusifs, c'est-à-dire qu'un observateur ne peut pas être simultanément dans deux de ces modes.

Requête/Réponse

Le premier mode de sécurité est le mode **requête/réponse**. Ce mode est illustré par la figure 3.6. Classiquement, lorsque l'observateur reçoit une requête, il va fournir une réponse. Dans ce mode, l'observateur peut refuser l'appel, retranscrit sur le schéma par les pointillés. Il observe l'appel système et transmet la réponse de l'appel au système. On peut ajouter un comportement facultatif à ce mode qui est l'émission d'une alerte.

Classiquement, ce mode est le **mode protection** (ou *enforcing*) que l'on retrouve dans les implantations telles que SELinux ou grsecurity. Lorsque l'observateur reçoit une requête, il autorise ou non l'interaction et répond à l'appelant (c'est-à-dire : l'observateur se comporte comme un

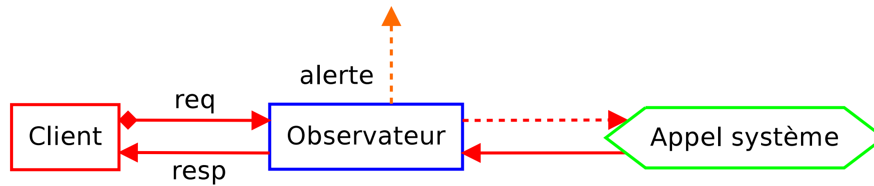

FIGURE 3.6 – Mode de sécurité : requête/réponse de l'observateur

intermédiaire entre le processus et l'appel système). Si l'interaction est refusée, une alerte sera enregistrée dans un fichier d'audit.

Évidence

Le second mode de sécurité est le mode **évidence**. Ce mode est illustré par la figure 3.7. Le fonctionnement se base sur deux étapes : lorsque l'observateur reçoit une requête, il transmet la requête à l'appel système et peut éventuellement émettre une alerte. Dans ce mode de fonctionnement, l'observateur exécute toujours l'appel système.

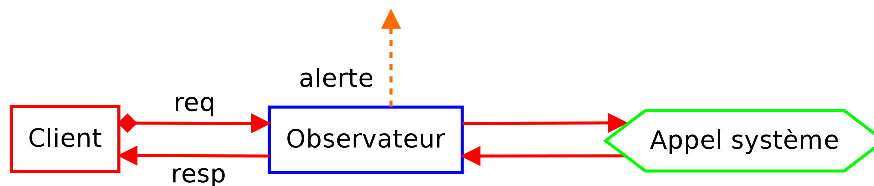

FIGURE 3.7 – Mode de sécurité : évidence

On retrouve ce comportement dans les systèmes de détection d'intrusion (*Intrusion Detection System*). Lorsque l'observateur reçoit une requête qui correspond à une des règles, il va lever une alerte (ou générer une trace), sinon il ne fait rien. On retrouve aussi ce comportement pour les mécanismes de protection qui sont placés en mode **détection** aussi appelé permissif.

Notification

Le troisième mode de sécurité est le mode **notification**. Ce mode est illustré par la figure 3.8. Le fonctionnement de ce mode repose sur le fait que l'observateur va envoyer une notification sans avoir reçu d'évènement spécifique.

FIGURE 3.8 – Mode de sécurité : notification

Il peut s'agir d'un observateur qui par exemple envoie un signal à un processus qui consomme trop de temps processeur.

3.1.2 Définition d'un observateur : le moniteur de référence

Nous allons, dans cette partie, décrire plus en détail notre modèle de moniteur de référence. Ce modèle est assez générique pour être appliqué sur les différents systèmes. Le moniteur de référence est chargé d'appliquer le traitement présent dans la politique de sécurité au regard de l'interaction courante. Dans le cadre de notre moniteur de référence, les traitements possibles sont : **autoriser**, **refuser** et **auditer**. Si ce modèle couvre les approches Unix existantes, nous montrerons qu'il peut être implanté sur différents systèmes. Nous proposons plus spécifiquement une mise en œuvre pour Windows puisque le manque se situe à ce niveau.

Le moniteur de référence est un observateur particulier qui peut empêcher l'interaction, il est donc en mode de sécurité **requête/réponse**. Cela se traduit de la manière suivante :

- la **requête** : c'est une demande d'accès d'un sujet sur un objet du système ;
- la **réponse** : c'est la décision du moniteur de référence pour la demande d'accès. Elle peut être de deux types : autorisation ou refus.

À la différence des moniteurs existants sur Linux, notre moniteur de référence est plus générique en supportant des politiques de sécurité de type PBAC et des politiques DTE, que nous détaillerons par la suite.

Notre moniteur de référence propose 3 modes d'utilisation :

- *désactivé* : aucun contrôle n'est réalisé et le flux d'exécution n'est pas détourné ;
- *actif* : le flux d'exécution est détourné et les décisions permettent d'empêcher les interactions. En cas de refus, une alerte est levée par la génération d'une trace. C'est le mode protection ;
- *permissif* : le flux d'exécution est détourné, le moniteur de référence cherche dans la politique et prend une décision, mais il n'empêche pas l'interaction. Ce mode permet de créer plus facilement la politique de contrôle d'accès et de la valider sans nuire au fonctionnement du système. Dans ce mode, par défaut, tous les accès, qu'ils soient autorisés ou non, peuvent être enregistrés dans un fichier d'audit. C'est le mode évidence.

La figure 3.9 illustre les différents traitements que réalise notre moniteur de référence. Pour décrire cette architecture logicielle, nous allons dérouler un cas pratique en détaillant chacune des étapes réalisées au sein du moniteur de référence. Nous séparerons en trois parties ce cheminement : les opérations réalisées durant une phase de pré-décision, c'est-à-dire avant de prendre une décision pour savoir si l'appel système sera exécuté ou non, une phase de décision, et les opérations réalisées en post-décision, c'est-à-dire une fois que la décision a été prise.

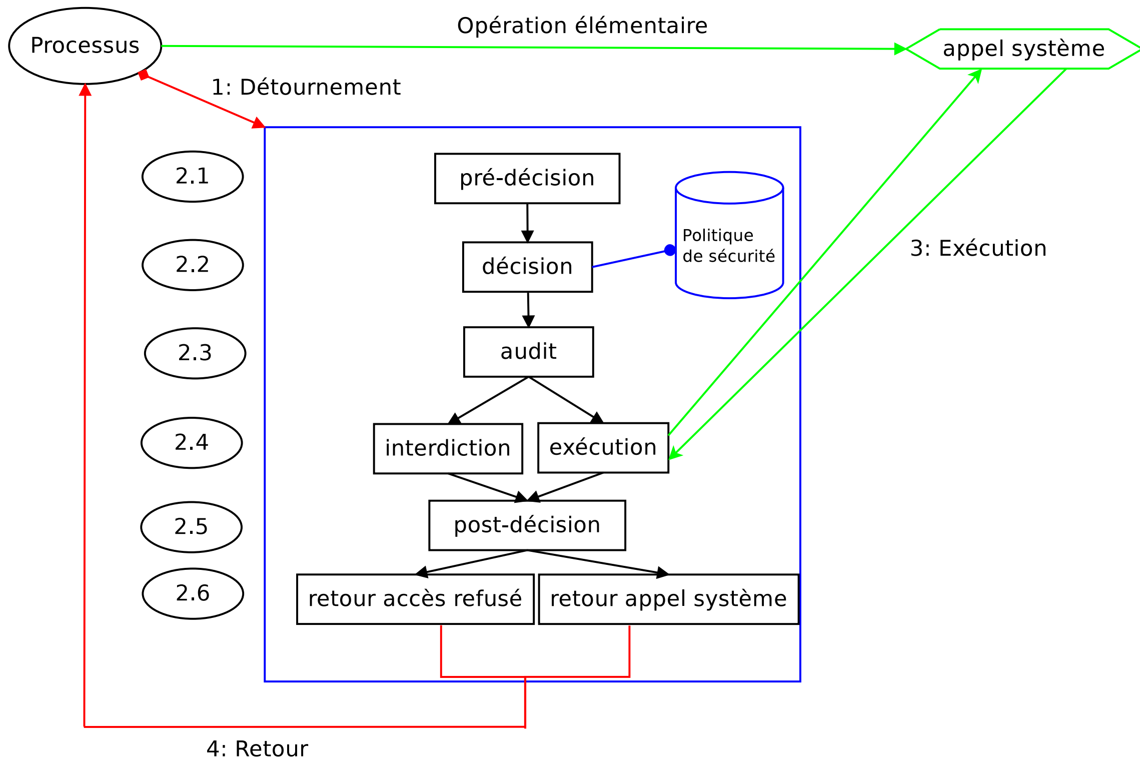

FIGURE 3.9 – Schéma du moniteur de référence

3.1.2.1 Phase de pré-décision

La première opération, qui n'est pas directement du ressort du moniteur de référence et qui consiste à détourner le flux d'exécution régulier, correspond à la flèche 1. Nous décrirons différentes méthodes de détournement pour les systèmes Windows dans la section 3.3. Cette partie agit comme les LSM et propose des interfaces de connexion à notre moniteur de référence. Ainsi, ce dernier n'a pas besoin de savoir comment est réalisé le détournement du flux d'exécution.

Prétraitement Une première phase notée 2.1 regroupe un ensemble d'opérations de **prétraitement**. Dans cette phase, le moniteur attribue un identifiant unique de trace et ajoute un horodatage. Nous simplifierons les notations en considérant que l'élément d contient à la fois l'horodatage et un identifiant unique. Ces deux informations serviront à reconstruire l'activité globale du système pour trouver les scénarios complets d'attaque. La figure 3.10 illustre ce mécanisme.

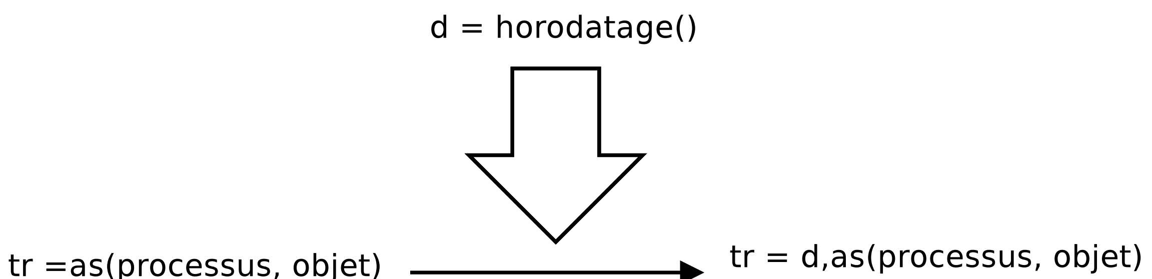

FIGURE 3.10 – Construction de la trace par le moniteur de référence : ajout de l'horodatage

Le moniteur ajoute ainsi des informations qui ne serviront pas directement à la prise de décision mais qui se révèlent utiles pour construire/analyser les activités du système. Dans notre cas, nous avons choisi de prendre les PID et PPID du sujet, illustré par la figure 3.11.

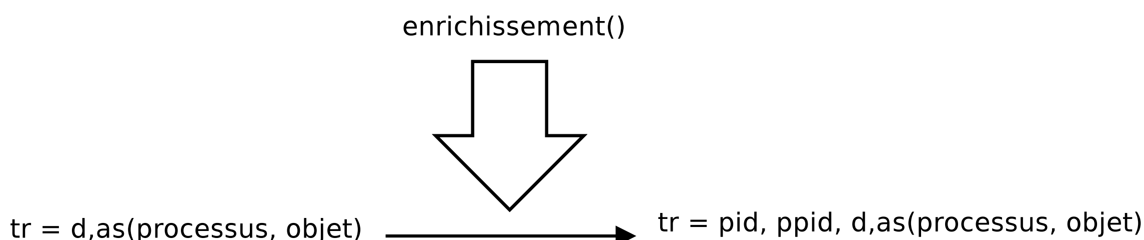

FIGURE 3.11 – Construction de la trace par le moniteur de référence : ajout d'information pour l'historique des processus

Notre moniteur de référence doit ensuite calculer les éléments qui vont lui servir à la prise de décision. Pour cela, il doit récupérer le sujet, l'objet, la classe de l'objet et l'opération élémentaire demandée par le sujet sur l'objet, illustré par la figure 3.12.

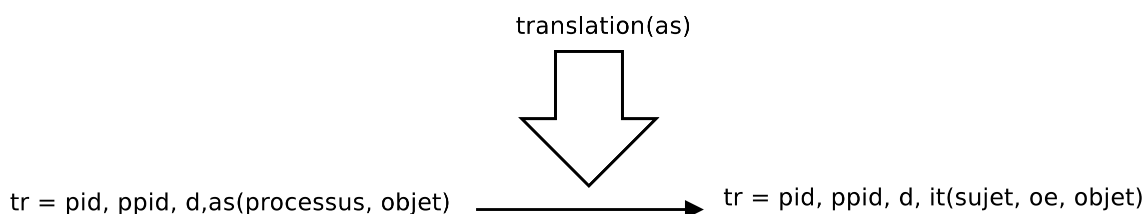

FIGURE 3.12 – Construction de la trace par le moniteur de référence : translation en une interaction

Nous calculons dans cette première phase l'interaction requise qui permettra au moniteur d'autoriser ou de refuser l'appel système.

Dans le cas du modèle de protection PBAC, le moniteur translate les sujets et objets vers leurs noms absolus (voir les détails ultérieurs). Les opérations élémentaires sont calculées sous la forme d'ACL étendues comme définies dans la politique.

Pour le modèle de protection DTE, il est nécessaire d'obtenir les contextes de sécurité incluant les types pour les sujets et objets. Ils sont donc calculés dans cette phase. Les opérations élémentaires sont elles directement dérivées des appels système.

A la fin de cette phase, le moniteur possède une interaction it décrivant l'interaction à contrôler.

Tout le cheminement de la construction de la trace effectuée en phase de prétraitement est illustré par la figure 3.13.

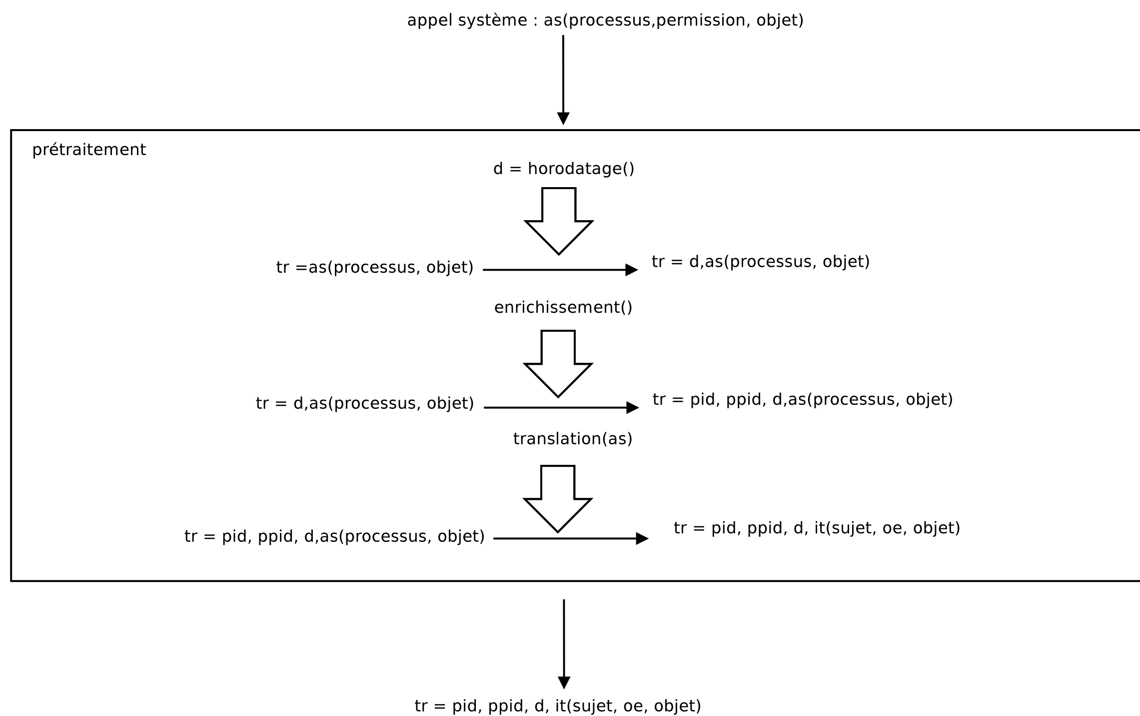

FIGURE 3.13 – Construction de la trace par le moniteur de référence : schéma global

L'algorithme de construction de la trace par le moniteur de référence est défini par l'algorithme 4. Il prend en entrée l'appel système composé d'un processus et d'un objet. Il commence par horodater la trace puis l'enrichit en ajoutant les PID et PPID. Enfin, il transforme l'appel système en une interaction directe.

Algorithm 4 Construction de la trace par le moniteur de référence

Require: $AS : processusobjet$

$tr \leftarrow processus, objet$

$d \leftarrow horodatage()$

$tr \leftarrow d, tr$

$pid, ppid \leftarrow enrichissement()$

$tr \leftarrow pid, ppid, tr$

$it \leftarrow translation(as)$

$tr \leftarrow modification_trace(tr, it)$

return tr

L'algorithme de translation de l'appel système en une interaction directe est illustré par l'algorithme 5. Il prend en entrée l'appel système et commence par modifier le processus et l'objet pour coller au modèle de protection (PBAC ou DTE). Puis, il va faire correspondre l'appel système à une opération élémentaire pour construire une interaction directe. La fonction $ObtenirOE()$ retourne l'opération élémentaire associée à l'appel système.

Algorithm 5 Translation de l'appel système en interaction

Require: $AS : processusobjet$
  $s \leftarrow Sujet(processus)$
  $o \leftarrow Objet(objet)$
  $oe \leftarrow ObtenirOE(AS)$
  $it \leftarrow (s, oe, o)$
  return it

3.1.2.2 Phase de décision

Une fois que le moniteur de référence a toutes les informations nécessaires, il est capable de prendre une décision, qui correspond à la phase 2.2 de notre schéma. Pour ce faire, il va aller rechercher dans la politique de sécurité l'interaction requise.

1. Si l'interaction ne satisfait pas la politique, alors le moniteur considère que l'appel système doit être refusé et qu'il faut auditer ce refus.
2. Si l'interaction est présente dans la politique, alors le moniteur effectue le traitement correspondant.

Le moniteur génère une trace lors des interdictions dans le but d'assurer la traçabilité des opérations effectuées sur le système. Lorsque les opérations sont autorisées par la politique de sécurité, il n'est pas nécessaire de générer de trace.

3.1.2.3 Phase de post-décision

audit Après la phase de décision, le moniteur entre dans la phase d'audit. Dans cette phase, il va pouvoir enregistrer les éléments de l'interaction : sujet, objet, opération élémentaire. De plus, nous avons récupéré lors de la phase de prétraitement une date et un identifiant unique, ainsi que des informations secondaires. Nous allons nous servir de tous ces éléments pour construire une trace suffisamment précise pour décrire l'interaction.

Cette phase est obligatoire lorsque le moniteur est mis en mode *permissif*.

détection et protection Lorsque le moniteur est en mode *permissif*, l'opération élémentaire se poursuit c'est-à-dire que l'appel système est toujours exécuté.

Lorsque le moniteur est en mode *actif*, en cas de refus explicite, l'interaction ne se poursuit pas et l'appel système n'est pas exécuté. Si l'interaction est autorisée, alors le moniteur va exécuter l'appel système (flèche 3). Il s'agit du mode protection.

post-traitement La phase de **post-traitement** permet d'effectuer des traitements après l'exécution de l'appel système ou en cas de refus.

Dans le cadre du retour de l'appel système, le moniteur peut mettre à jour la trace d'audit en spécifiant par exemple le code de retour de l'appel système : s'il a réussi, échoué, quel code d'erreur, etc. La mise à jour de la trace passe aussi par l'inscription de la date de retour de l'appel système. Ainsi, en traitant les différentes traces, on peut connaître les sujets ayant eu accès en même temps à une ressource et ainsi contrer les attaques de type *race condition*, en analysant les dépendances causales entre interactions [Rouzaud-Cornabas, 2010].

Lorsque l'interaction a été refusée, le moniteur peut notifier les utilisateurs en fonction de la gravité de l'opération élémentaire. Par exemple, il peut envoyer un message directement depuis l'espace noyau pour déclencher une alerte.

retour du moniteur La dernière phase consiste à retourner le résultat de l'appel système au processus appelant. Lorsque l'interaction a été autorisée et que l'appel système a été exécuté, on peut renvoyer les résultats retournés par l'appel système. Lorsque l'interaction a été refusée, il faut renvoyer un code d'erreur significatif au processus. Par exemple sous Windows, le moniteur peut renvoyer le code `STATUS_ACCESS_DENIED` et sous Linux `-EACCES`. Il est à noter que le post-traitement permet de modifier si besoin les résultats de l'appel.

3.1.3 Répartition des observateurs

Nous avons présenté dans l'introduction de notre étude 1.3.3 qu'un seul observateur n'était pas suffisant pour protéger efficacement le système et qu'il était nécessaire de mettre en place plusieurs observateurs.

La répartition des observateurs consiste à ajouter différents observateurs dans la chaîne de contrôle des interactions, c'est-à-dire des appels système. Définir la répartition nécessite de décrire la méthode d'association de deux observateurs, leurs localisations et leurs redondances.

3.1.3.1 Association

Nous définissions deux modes d'association pour qu'une interaction puisse être traitée par deux observateurs. Ces deux modes sont mutuellement exclusifs. Deux observateurs ne peuvent pas être à la fois en mode cascade et en mode continuation, puisque comme nous le verrons, dans le premier mode, le premier observateur attend la réponse du second, ce qui n'est pas le cas dans le second mode.

Cascade

Le premier mode est le mode **cascade**. Ce mode est illustré par la figure 3.14. Dans ce mode, le premier observateur reçoit la requête et la transmet au second observateur. Le second observateur renvoie la réponse au premier observateur qui renvoie ensuite une réponse à l'appelant.

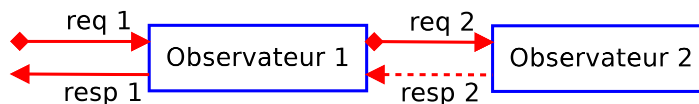

FIGURE 3.14 – Mode d'association : cascade

Ce mode de fonctionnement a été, par exemple, mis en place dans le défi ANR [ANR, 2009] par l'utilisation combinée de SELinux et de PIGA. Il permet d'additionner les contrôles des deux observateurs.

Continuation

Le second mode est le mode **continuation**. Ce mode est illustré par la figure 3.15. Dans le mode continuation, le premier observateur n'attend pas la réponse du second. Il effectue son traitement et transmet ensuite, si nécessaire, la requête au second.

Ce mode peut être illustré en donnant une priorité dans les contrôles d'accès. Le premier contrôle est prioritaire et décide seul d'interdire/autoriser l'accès ou bien de transmettre la demande au second mécanisme.

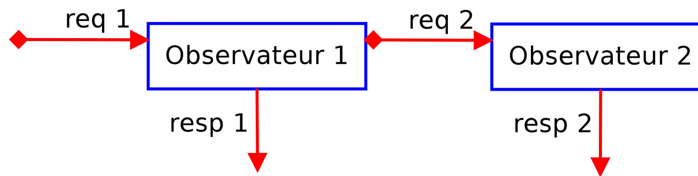

FIGURE 3.15 – Mode d'association : continuation

3.1.3.2 Localisation

Nous avons défini les modes de sécurité ainsi que les modes d'association possibles pour les observateurs. Nous allons maintenant détailler les modes de localisation, c'est-à-dire la place d'un observateur au sein de différentes architectures.

Colocalisée

Le premier mode concerne un observateur localisé sur la même machine que le client. Ce mode est illustré par la figure 3.16.

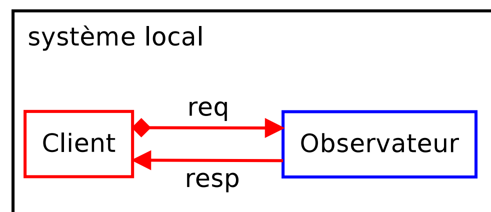

FIGURE 3.16 – Mode de localisation : colocalisé

Ce mode est principalement utilisé par les mécanismes de contrôle d'accès obligatoire : SELinux, grsecurity, etc.

Distante

Le second mode de répartition possible est d'avoir un observateur **distant**. Ce mode est illustré par la figure 3.17. Dans ce mode, les requêtes ne sont pas transmises localement, mais envoyées depuis un client vers un nœud *distant*. Nous avons dans ce schéma une relation 1 : 1, c'est-à-dire qu'à chaque nœud, nous associons un observateur spécifique sur une machine distante.

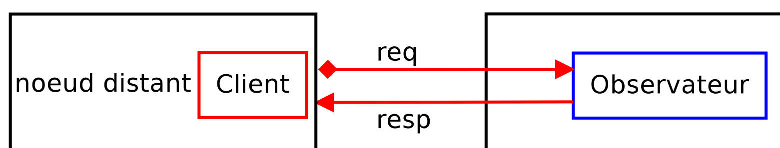

FIGURE 3.17 – Mode de localisation : distant

Ce mode peut être illustré par le fonctionnement en mode client/serveur du programme de gestion des traces système (`syslog-ng`). Le serveur de centralisation des logs reçoit les logs depuis tous les nœuds distants.

Parallèle

Ce troisième mode de répartition est différent du précédent. Le mode **distant** associe à chaque nœud une machine hébergeant un observateur, ce qui pose clairement un problème de dimensionnement du système réparti. Dans le mode **parallèle**, une même machine est capable d'héberger plusieurs observateurs qui s'exécutent en parallèle. Cependant, chaque observateur n'est associé qu'à un seul nœud distant. Ce mode est illustré par la figure 3.18. S'il est difficile de qualifier ce modèle, on peut le noter $n : 1(||)$ au sens que n nœuds peuvent être traités en parallèle par 1 machine.

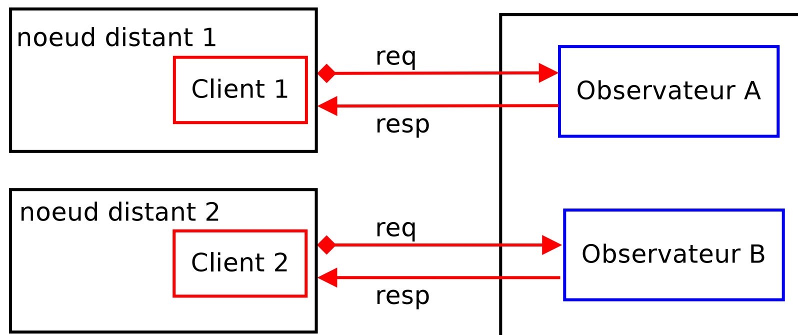

FIGURE 3.18 – Mode de localisation : distant avec plusieurs nœuds et plusieurs observateurs

Partagée

Dans le mode **partagé**, au lieu d'avoir une relation $1 : 1$ entre les nœuds et les observateurs, la relation se note en $n : 1(p)$. Cela signifie qu'un observateur est partagé par plusieurs nœuds distants. Ce mode est illustré par la figure 3.19.

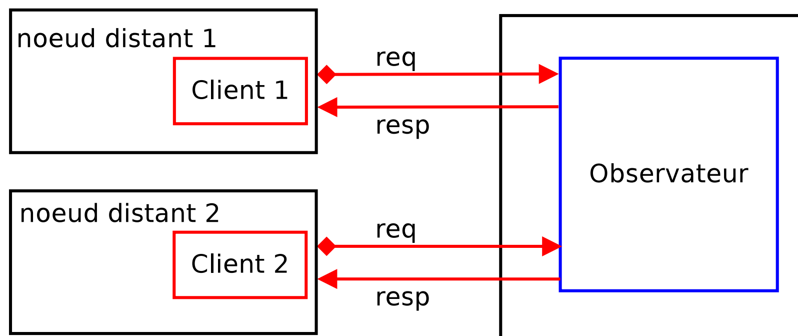

FIGURE 3.19 – Mode de localisation : distant avec plusieurs nœuds et un seul observateur

3.1.3.3 Redondance

Dans les architectures distribuées, il est nécessaire d'assurer la disponibilité des services. Dans le cas qui nous intéresse, l'observateur est un élément vital qu'il faut pouvoir redonder.

Diffusion

Le premier mode de redondance est le mode **diffusion**. Dans ce mode, le nœud envoie simultanément les requêtes à (au moins) deux observateurs. Il doit ainsi traiter les deux réponses des

observateurs. Ce mode est illustré par la figure 3.20. Ainsi, si un des deux observateurs ne répond plus, le nœud n'est pas obligatoirement bloqué puisqu'il peut obtenir une réponse du second observateur. Dans le cas de fautes byzantines, on peut imaginer au moins trois redondances (R1, R2, R3) avec un vote majoritaire.

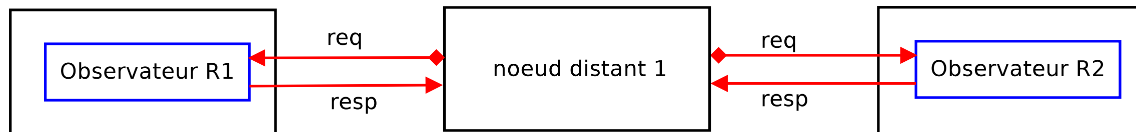

FIGURE 3.20 – Mode de redondance : diffusion des requêtes par le nœud

Maître-Esclave

Le second mode de redondance est le mode **maître-esclave**. Dans ce mode, ce sont les observateurs qui se communiquent les requêtes envoyées par le nœud au serveur maître. Nous avons pris comme exemple l'observateur maître qui fait suivre la requête. Ainsi, les deux observateurs possèdent tous les deux toutes les requêtes envoyées. Ce mode de fonctionnement est illustré par la figure 3.21. Lorsque l'observateur maître ne répond plus, le nœud se connecte à l'observateur esclave.

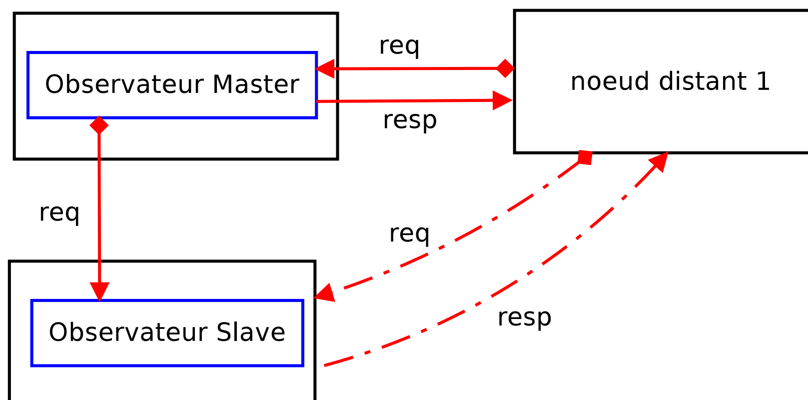

FIGURE 3.21 – Mode de redondance : architecture de la forme maître-esclave

3.1.3.4 Discussion

Nous venons de détailler les différents modes d'association, de localisation et de redondance des observateurs.

Par la définition des modes d'association, nous modélisons l'utilisation combinée de deux observateurs. Dans le mode en cascade, le premier observateur a un rôle primordial dans la chaîne de traitement des requêtes puisque c'est lui qui doit envoyer les requêtes au second observateur et attendre la réponse. Il est ainsi possible de corréler, au niveau du premier observateur, les décisions prises par les deux observateurs. Cependant, dans ce mode de fonctionnement, le premier observateur doit attendre une réponse du second observateur, ce qui peut réduire les performances du système. Dans le second mode, le premier observateur n'attend pas la réponse du second. Il effectue son traitement et transmet ensuite, si nécessaire, la requête au second. Ce mode de fonctionnement évite le blocage du premier observateur attendant la réponse du second. Cependant, il n'est plus possible de corréler les réponses des observateurs au niveau du premier.

Les modes de localisation adaptent les modèles des systèmes répartis aux observateurs. À partir du mode distant, nous avons spécialisé l'approche de répartition. Ainsi, par la définition des modes parallèle et partagé, nous avons centralisé les requêtes et les décisions prises par les observateurs. Avec une localisation distante et les deux cas particuliers (parallèle et partagé), il semble possible de réduire la charge de calcul des nœuds clients. De plus, avec les modes parallèles et partagés, il est possible de faire des corrélations entre les interactions des différents nœuds clients.

Néanmoins, en déportant les observateurs sur une machine dédiée, nous introduisons des situations de pannes supplémentaires : les pannes sur le réseau et les pannes de la machine hébergeant l'observateur déporté. De plus, ce déport peut aussi entraîner une latence au niveau de la communication avec l'observateur puisque cette communication passe par un lien réseau.

Pour réduire les pannes potentielles introduites par le déport de l'observateur, nous avons proposé un modèle de redondance. Nous considérons deux modes de fonctionnement. Le premier repose sur la diffusion des requêtes faites directement par le nœud. Le principal avantage de cette méthode est qu'elle permet de tolérer un grand nombre de situations de défaillance. Cependant, le nœud doit gérer les réponses envoyées par plusieurs observateurs, ce qui implique un calcul supplémentaire réalisé au niveau du nœud. Le second mode de fonctionnement propose un modèle maître-esclave. Dans ce mode de redondance, le nœud envoie les requêtes qu'à un seul observateur, le maître et c'est ce dernier qui transmet à l'observateur esclave. Cependant, lorsque l'observateur maître ne répond plus, c'est au nœud client de se connecter à l'observateur esclave, ce qui peut entraîner une latence importante, voir un blocage du nœud si l'observateur met trop de temps à répondre.

Nous venons de donner une définition de la notion d'observateur pour les systèmes et une modélisation de la notion de la répartition des observateurs. Nous allons maintenant décrire la modélisation de la politique de sécurité sur laquelle se base l'observateur pour effectuer ces traitements.

3.2 Modélisation d'une politique de contrôle d'accès direct

Une politique de sécurité définit les traitements à entreprendre par l'observateur, tels que : autoriser, refuser, auditer, détecter, etc. Dans le cadre d'une politique de sécurité ayant pour objet de faire du contrôle d'accès direct, nous avons vu dans l'état de l'art (voir en section 2.2.2), qu'il existe plusieurs modèles obligatoires. Nous avons choisi ici deux modèles de contrôle d'accès direct : le *path-based access control* et le *domain and type enforcement*. Nous proposons une grammaire commune à ces deux modèles.

La création d'une politique de sécurité impose de nommer chaque ressource du système, que ce soit les processus, les fichiers ou les sockets par exemple. Nous allons expliquer pourquoi il est nécessaire de proposer un système de noms absolus indépendants de la localisation pour désigner les ressources. Grâce à ce système de nommage, nous pouvons générer des politiques de sécurité portables. Nous montrons l'usage pour les systèmes Windows. Nous définissons une grammaire commune supportant PBAC et DTE pour Windows. Nous présentons ensuite une étude des mécanismes de détournement pour Windows.

3.2.1 Grammaire de la politique

Une politique de protection regroupe l'ensemble des interactions qui sont autorisées ou interdites. Ces règles peuvent être :

— des règles de décision : autorise ou refuse les interactions ;
— des règles d'audit : définit si l'interaction doit être auditée ;

```
grammar gram_commune;

politique : (regle'\n')*;
regle : regle_traitement|regle_transition;

regle_traitement : regle_decision|regle_audit;
regle_transition   : role_transition|objet_transition;

regle_decision : autorise_interaction vecteur_acces;
regle_audit        : audit vecteur_acces;

role_transition : autorise_role ' ' role ' ' role;
objet_transition : autorise_objet ' ' contexte ' ' contexte (':' classe)?  ' ' contexte ;

vecteur_acces : contexte Delimiteurs_debut? (liste_operations_elementaires)+
    Delimiteurs_fin? ;
liste_operations_elementaires : (' '|'\t')* contexte(':' classe)? operations_elementaires
    ;
```

Listing 3.1 – grammaire commune aux modèles de protection PBAC et DTE

— des règles de transition : modifie l'état de la politique ou les contextes du système (sujet ou objet).

Nous avons défini une grammaire commune pour l'écriture d'une politique de contrôle d'accès, que l'on retrouve dans le listing 3.1.

Notons que cette grammaire ne propose aucun terminal. En effet, les terminaux sont propres à chaque modèle de protection. Nous spécifierons dans la suite de cette étude les terminaux pour les modèles PBAC et DTE.

Nous allons maintenant détailler la grammaire.

Les ressources du système (contextes) Nous avons distingué dans une interaction directe l'élément sujet, qui réalise l'opération élémentaire, de l'élément objet. Dans la grammaire, il n'est pas nécessaire de distinguer ces deux éléments puisque les sujets et les objets sont des sous-ensembles de l'ensemble des contextes. C'est pourquoi nous regroupons ces deux notions au sein d'un même ensemble que nous appelons **contexte**.

Définition 3.2.1 ***Un contexte*** *identifie une ressource à l'aide d'un nom. On retrouve dans la littérature essentiellement deux méthodes pour nommer les ressources d'un système. La première méthode consiste à utiliser le chemin complet dans l'arborescence du système de fichiers, ce qui est fait dans le modèle* PBAC *utilisé par grsecurity 2.3.1.2 tandis que la seconde méthode est d'associer un label à la ressource pour la caractériser, ce qui est fait dans le modèle* DTE*, comme le fait SELinux (voir en section 2.3.1.1).*

On notera CS l'ensemble des contextes sujet et CO l'ensemble des contextes objet.

La figure 3.22 illustre l'opération d'écriture faite par le processus `Firefox.exe`. Dans le modèle PBAC, ce dernier est représenté par son chemin complet `C:\Program Files\Mozilla\Firefox.exe`. Il réalise une écriture sur le fichier `C:\Users\bob\Downloads\malware.exe`.

Nous pouvons faire de même pour les systèmes Linux, illustré aussi par la figure 3.22.

La figure 3.23 illustre cette opération d'écriture d'un contexte sujet sur un contexte objet en utilisant le modèle DTE. Comme ce modèle associe des types ou des domaines aux ressources, nous associons au processus `Firefox.exe` le domaine `domaine_firefox` et au fichier `malware.exe` le type `type_malware`. Le principal avantage de ce modèle est la possibilité d'abstraire les ressources du système. Ainsi, il suffit d'associer le domaine `domaine_firefox`

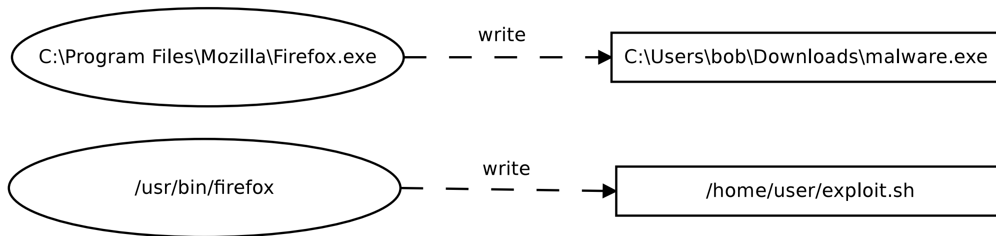

FIGURE 3.22 – Nommage des ressources au format PBAC

au processus `/usr/bin/firefox` et le type `type_malware` au fichier `exploit.sh` pour que la même figure illustre cette même opération sur un système Linux.

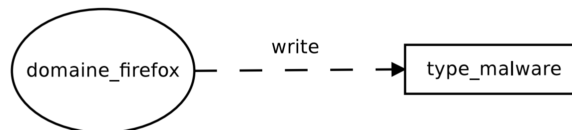

FIGURE 3.23 – Nommage des ressources au format DTE

Une politique de sécurité Une politique de sécurité est un ensemble de deux types de règles : les règles de traitement, qui définissent les actions de l'observateur vis-à-vis des interactions et les règles de transition, qui vont modifier l'état effectif de la politique.

Règle de traitement Une règle de traitement est composée de deux éléments : un traitement spécifique à effectuer par l'observateur et un vecteur d'accès. Le tableau de vecteurs d'accès détermine les sujets, objets et les opérations élémentaires de l'interaction. Le traitement de l'observateur peut être : une autorisation explicite, un audit, une interdiction explicite, etc. Le vecteur d'accès détermine des opérations élémentaires entre un contexte sujet et un contexte objet.

Règle de transition Une règle de transition va modifier l'état effectif de la politique ainsi que les contextes des sujets et des objets. Nous définissons deux règles de transition :

1. une transition entre deux rôles ;
2. une transition entre différentes ressources du système.

La transition entre deux rôles permet de modifier l'ensemble des opérations élémentaires accessibles. Par exemple, un utilisateur ayant le rôle **user** aura besoin de changer de rôle pour effectuer des tâches d'administration. La figure 3.24 illustre les interactions autorisées pour les rôles **user** et **admin**.

Les transitions entre les ressources du système interviennent lors de la création d'un processus. Dans ce cas, la transition provient du chargement du fichier en mémoire dans le but de créer un nouveau processus.

Vecteur d'accès Un vecteur d'accès v représente un ensemble d'interactions directes à observer sur le système. Il est composé d'un contexte sujet, d'un contexte objet et d'une liste d'opérations élémentaires potentiellement regroupées par un ensemble associé à une classe.

Soit $cs \in CS$, soit $co \in CO$, soit $class \in Class$ et $oe_1, ..., oe_n \in OE$,

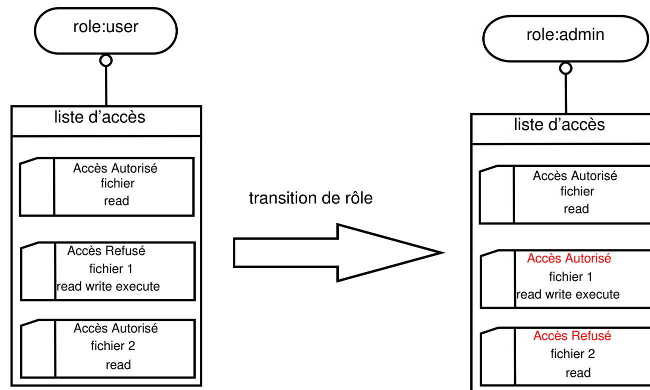

FIGURE 3.24 – Transition de rôle modifiant les accès possibles à certaines ressources du système

$$\text{alors } v = (cs, co : class, \{oe_1, ..., oe_n\})$$

Selon la figure 3.2 (présente au début de ce chapitre), pour une seule interaction sur un système Windows entre le contexte sujet `firefox.exe` et le contexte objet `malware.exe` on peut par exemple écrire un vecteur :

$$v = (firefox.exe, malware.exe : file, \{write\})$$

On peut factoriser un ensemble d'interactions entre deux contextes avec un seul vecteur de la façon suivante :

$$v = (firefox.exe, malware.exe : file, \{create, setattr\})$$

Cependant, nous verrons qu'il est préférable d'avoir des contextes précis (ici la règle s'applique à tous les sujets `firefox.exe` présents sur le système ce qui peut être acceptable pour une interdiction mais peu recommandé pour une autorisation) avec dans ce cas la nécessité que les noms soient indépendants de leurs localisations physiques afin de garantir la portabilité des politiques.

3.2.1.1 La portabilité des noms utilisés pour les contextes

Dans notre grammaire, un contexte est un nom qui désigne aussi bien les sujets et les objets. Il est donc nécessaire que ces noms soient précis afin de contrôler finement chaque processus et chaque ressource du système. En même temps, ces noms doivent être les plus portables possibles.

Le système de nommage sous Windows

Sur les systèmes Windows, il n'existe pas de racine commune à l'espace de noms. Historiquement, les systèmes Windows sont installés sur la lettre du lecteur "`C:`" car les lecteurs "*A :*" et "*B :*" étaient utilisés pour les lecteurs de disquettes. Il est cependant possible d'installer le système sur un autre lecteur mais aussi d'avoir plusieurs partitions montées sur des lecteurs différents. Ainsi, le listing 3.2 montre sur sa première ligne le chemin classique pour l'exécutable de `Firefox` alors que la seconde ligne montre un chemin depuis un autre lecteur pour le même exécutable.

```
1 C:\Program Files\Mozilla\Firefox.exe
2 D:\Program Files\Mozilla\Firefox.exe
```

Listing 3.2 – Deux chemins Windows différents désignant le même exécutable

Si l'on utilise le nom `C:\Program Files\Mozilla\Firefox.exe` désignant précisément l'exécutable `Firefox` concerné, ce nom n'est pas portable puisqu'il n'est pas correct sur un système où les programmes sont installés sur le système de fichiers *D :*.

Pour résoudre ce problème, nous utilisons la notion de `nom symbolique absolu indépendant de la localisation` définie par [Hagimont et J.Mossière, 1996].

1. Un nom symbolique, par exemple le fichier `./policy.24`, correspond à un nom usuel qui est manipulé par les usagers. Ce nom usuel est associé à une localisation physique, sans que l'utilisateur n'ait besoin de la connaître. Ainsi le nom symbolique `./policy.24` est par exemple associé au numéro d'*i-node* 11274188 et au périphérique `19h/25d` sous Unix.
2. Un nom absolu rend le nom symbolique unique c'est-à-dire qu'il désigne une seule ressource. Le nom absolu est par exemple `/home/damien/policy.24` en opposition au nom relatif `policy.24`.
3. Un nom absolu indépendant de la localisation physique est un nom qui ne fait pas apparaître l'emplacement de stockage de la ressource. Cette solution est particulièrement efficace pour assurer la portabilité des noms.

Pour atteindre ce but sur les systèmes Windows, nous allons abstraire le nom usuel pour rendre absolu et indépendant de la localisation le nom symbolique.

Définitions

Pour formaliser cette problématique de nommage sur Linux et sur Windows, nous allons définir les deux notions importantes que sont **nom** et **ressource**.

Définition 3.2.2 ***Ressource*** *Une ressource est un identifiant interne au système utilisé pour manipuler des objets : par exemple, l'*i-node *d'un système de fichier.*

Définition 3.2.3 ***Nom*** *Un nom est un moyen simple et usuel pour accéder à une ressource. Généralement, le nom est représenté sous la forme d'une chaîne de caractères.*

Ainsi, une même ressource peut avoir plusieurs noms différents. Nous illustrerons cela dans la suite de cette partie.

3.2.1.2 Modélisation de la problématique des noms

Nous allons maintenant montrer comment fonctionne le mécanisme de nom symbolique absolu indépendant de la localisation. Notre modélisation commence par définir deux éléments importants du système : les notions d'**Emplacement physique** et de **Ressource**.

Définition 3.2.4 ***Emplacement physique*** *Soit e_{phy} un élément de l'ensemble E_{phy} regroupant les emplacements physiques. Concrètement, e_{phy} correspond à un bloc de données sur le disque dur.*

Définition 3.2.5 ***Ressource*** *Soit r une ressource du système, par exemple un fichier, appartenant à l'ensemble R, l'ensemble de toutes les ressources du système. Cette ressource r se caractérise par un ensemble d'emplacement physique. Nous noterons RSC la fonction qui associe à une ressource r un ensemble d'emplacement physiques.*

$$\forall r \in R, RSC\ r \mapsto e \text{ tel que } e = \{e_{phy1}, .., e_{pyhn}\} \text{ avec } \{e_{phy1}, .., e_{pyhn}\} \in E_{phy}$$

Toute ressource du système possède, au moins, un élément dans l'espace d'arrivée E_{phy}. A contrario, tout élément de l'espace E_{phy} ne possède pas nécessairement un antécédent dans l'espace des ressources. Par exemple, lorsqu'un fichier est supprimé par le système par les fonctions classiques de suppression telles que `rm`, les données présentes sur le disque dur associées à ce fichier ne sont pas supprimées. Il faut, pour cela, utiliser des fonctions spécifiques.

De plus, tout élément de E_{phy} qui possède un antécédent, ne possède qu'un seul et unique antécédent par la fonction RSC. Donc par définition, la fonction RSC est injective. Ceci est illustré par le schéma 3.25.

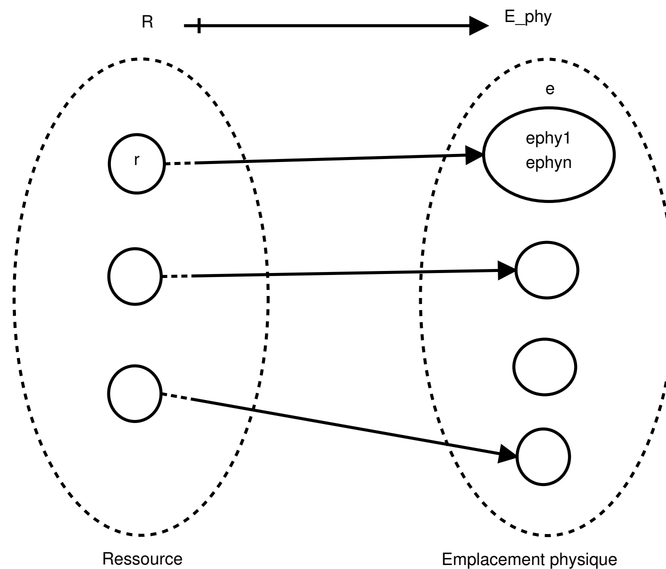

FIGURE 3.25 – Correspondance entre une ressource et les emplacements physiques

En outre, chaque ressource du système possède un emplacement logique. Cet emplacement logique se caractérise sur le système par un chemin dans le système de fichiers. Une ressource peut posséder plusieurs emplacements logiques différents, grâce à des opérations de montage ou par la création de liens.

Définition 3.2.6 ***Emplacement logique et chemins*** *Nous noterons E_{log} la fonction qui associe à une ressource* r *un emplacement logique dans le système. Cette fonction renvoie un chemin complet de l'arborescence du système de fichiers. L'ensemble de tous les chemins complets possibles est noté $Chemin$. C'est cet emplacement logique qui est utilisé par l'utilisateur pour accéder à la ressource.*

$$\forall r \in R, E_{log} :\ r \mapsto chemin \text{ avec } chemin \in Chemin$$

Pour qu'une ressource du système puisse être utilisée par un utilisateur, il est nécessaire qu'elle possède au moins un emplacement logique, c'est-à-dire un nom dans l'espace des noms logiques. Une ressource peut posséder plusieurs emplacements logiques grâce à des opérations de montage ou par la création de lien. Cette notion est illustrée par la figure 3.26. On obtient pour un fichier, une ressource correspond à l'*i-node* et un ensemble d'emplacement physique.

Comme une ressource peut être désignée par plusieurs emplacements logiques, il est nécessaire de définir une nouvelle fonction capable d'uniformiser la description d'une ressource pour un utilisateur.

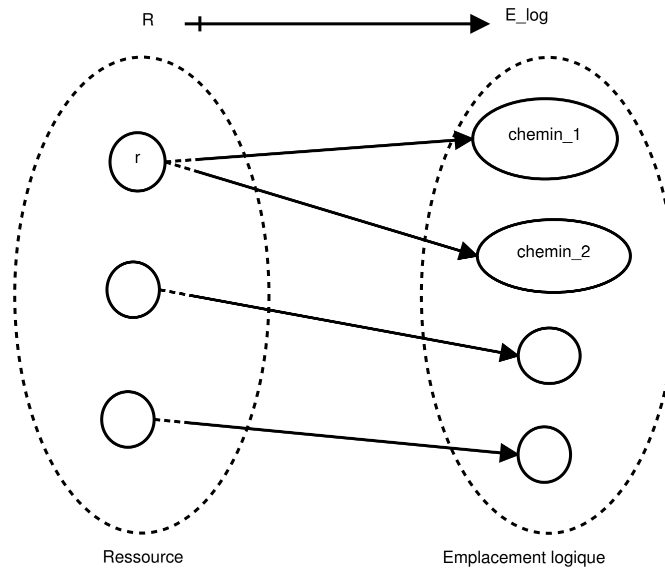

FIGURE 3.26 – Correspondance entre une ressource et les emplacements logiques

Définition 3.2.7 ***Nom symbolique absolu indépendant de la localisation*** *Nous définissons la fonction N_{abs}. Cette fonction associe à un emplacement logique d'une ressource un nom symbolique absolu indépendant de la localisation $n_r^{abs} \in N^{abs}$, où N^{abs} est l'ensemble des noms symboliques absolus indépendants de la localisation.*

Soit $r \in R$ et soit $chemin \in Chemin$ avec $E_{log} : r \mapsto chemin$, alors $N_{abs} : chemin \mapsto n_r^{abs}$

Par définition, tout emplacement logique possède un seul et unique nom symbolique absolu indépendant de la localisation. A contrario, un nom symbolique absolu indépendant de la localisation possède au moins un emplacement logique. La fonction N_{abs} est donc surjective, illustrée par la figure 3.27.

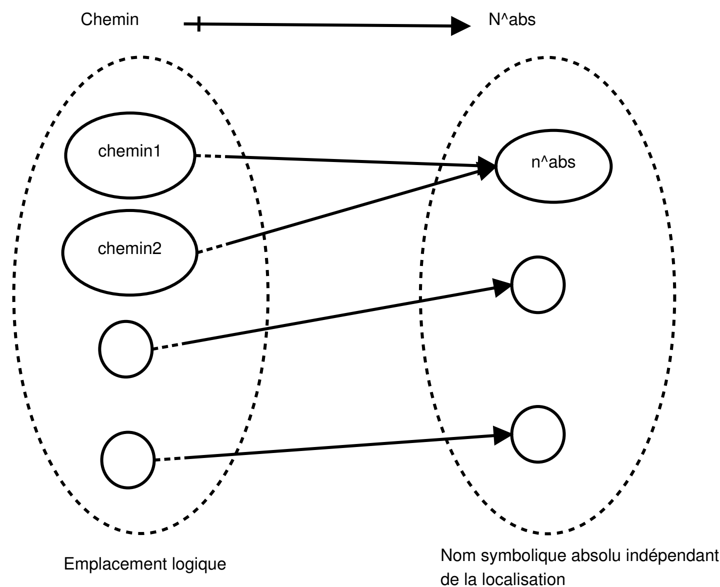

FIGURE 3.27 – Représentation de la fonction N_{abs}

La propriété essentielle de cette fonction est la suivante : si une même ressource possède différents emplacements logiques, alors, par la fonction N_{abs}, elle ne possède qu'un seul et unique nom symbolique absolu indépendant de la localisation. Cette propriété s'exprime de la façon suivante :

$$\forall r \in R,\ \text{si } \exists E_{log}(r) = chemin_1 \text{ et } E_{log}(r) = chemin_2$$

$$\text{avec } chemin_1 \neq chemin_2 \Rightarrow N_{abs}(chemin_1) = N_{abs}(chemin_2)$$

Cette propriété sera illustrée par la suite.

Solution pour les systèmes Windows

Sur les systèmes Windows, la problématique de nommage des ressources est essentiellement due au fait qu'il n'y a pas de racine commune entre les différents systèmes. Pour résoudre ce problème et pour appliquer notre fonction permettant d'obtenir un nom symbolique absolu indépendant de la localisation, nous allons utiliser un mécanisme interne au système commun à chaque système Windows.

Dans un premier temps, le nom de la ressource est séparé en deux : la lettre du lecteur qui est potentiellement changeante, et son chemin dans l'arborescence qui est constant. Nous appliquons la fonction N_{abs} à la partie changeante. Pour ce faire, nous utilisons la notion d'**espace de noms** (*namespace*) d'une ressource. L'espace de noms d'une ressource est une fonctionnalité du système qui associe aux ressources une représentation abstraite indépendante de leur localisation.

La fonction N_{abs} est appliquée à `C:\Program Files` exemple présenté dans le listing 3.2. C'est la partie que l'on considère comme changeante. On obtient ainsi $N_{abs}(C : \backslash ProgramFiles) = ns_programfiles$.

En reprenant notre exemple présenté dans le listing 3.2, nous obtenons pour les deux noms usuels différents 3.3 un même nom symbolique absolu indépendant de la localisation.

```
1 C:\Program Files\Mozilla\Firefox.exe : ns_programfiles:\Mozilla\Firefox.exe
2 D:\Program Files\Mozilla\Firefox.exe : ns_programfiles:\Mozilla\Firefox.exe
```

Listing 3.3 – Application des noms symboliques absolus indépendants de la localisation pour identifier une même ressource sur un système Windows

L'utilisation de cet espace de noms offre une meilleure portabilité de la politique de contrôle d'accès. Cet espace de noms étant géré directement au sein du système Windows, il n'est plus nécessaire de réécrire les politiques pour les appliquer sur un autre système Windows.

Solution pour les systèmes Linux

Les systèmes Linux utilisent des identifiants internes pour différencier les ressources sur un système. Même si ces identifiants sont accessibles depuis l'espace utilisateur, ils ne sont pas facilement manipulables. Et même si les systèmes Linux disposent d'une racine commune, nous pouvons aussi appliquer notre formalisation à ce système.

Par exemple en adaptant le listing 3.4 et en appliquant la fonction N_{abs}, nous obtenons le résultat suivant 3.5.

```
1 /lhome/gros/policy.24 : ns_home:gros/policy.24
2 /mnt/home/gros/policy.24 : ns_home:gros/policy.24
```

Listing 3.5 – Application des noms symboliques absolus indépendants de la localisation pour identifier une même ressource sur un système Linux

```
gros@ossus:~% stat /lhome/gros/policy.24
  File: '/lhome/gros/policy.24'
  Size: 1592659 Blocks: 3112 IO Block: 32768 regular file
Device: 19h/25d Inode: 11274188 Links: 1

sudo mount -o bind /home/ /mnt/

gros@ossus:~% stat /mnt/home/damien/policy.24
  File: '/mnt/home/gros/policy.24'
  Size: 1592659 Blocks: 3112 IO Block: 32768 regular file
Device: 19h/25d Inode: 11274188 Links: 1
```

Listing 3.4 – Différents noms Linux pour une même ressource identifiée par son numéro *i-node*

3.2.2 Application de la grammaire sur deux modèles de protection pour les systèmes Windows

L'étude des travaux précédents a montré qu'il existait des implantations de modèles de protection pour les systèmes Linux. C'est pourquoi nous allons nous concentrer exclusivement sur les systèmes Windows pour l'application de notre grammaire. La première partie de cette section a démontré que cette grammaire pouvait s'appliquer sur les deux systèmes d'exploitation.

Nous appliquons la grammaire que nous venons de définir sur deux modèles de protection : PBAC et DTE. Dans cette section, nous montrerons que seuls les terminaux de la grammaire doivent être spécifiés, mais que ces deux modèles se basent sur la grammaire commune que nous avons définie. Nous avons choisi le modèle PBAC pour la facilité qu'il permet dans la rédaction de la politique car ce modèle se base sur les chemins des ressources. Le second modèle, DTE, offre une couche d'abstraction entre la politique et le système permettant d'avoir des politiques plus précises mais aussi supportant mieux l'hétérogénéité.

3.2.2.1 Policy-Based Access Control

Représentation du système

Grâce à l'introduction de notre mécanisme d'abstraction des noms, il est possible de décrire complètement le système.

Nous définissons NS la fonction du système qui associe à un répertoire donné un élément de l'espace de noms. Par exemple, `NS(C:\ProgramFiles)=ns_programfiles` où `ns_programfiles` est un élément de l'espace de noms.

Les objets Les objets du système sont toutes les entités qui subissent les interactions : fichiers, répertoires, éléments du registre, mais aussi les processus.

Grâce à sa stabilité d'architecture, il existe un certain nombre de répertoires communs entre les différentes versions des systèmes Windows. Il est donc possible de définir des éléments propres à l'espace de noms pour ces répertoires. Le listing 3.6 fait la correspondance entre les répertoires et leur nom dans l'espace de noms.

```
NS(C:\Windows) = ns_windows
NS(C:\Program Files) = ns_programfiles
NS(C:\Users) = ns_users
NS(C:\Recycler) = ns_recycler
```

Listing 3.6 – Correspondance entre les répertoires système et leur nom dans l'espace de noms

```
NS(C:\Windows\explorer.exe) = ns_windows\explorer.exe
NS(System) = system
```

Listing 3.8 – Noms associés aux sujets pour le modèle PBAC

Lorsqu'un répertoire est spécifique à un système, ou simplement ajouté par un administrateur, il est possible de lui ajouter un élément générique de l'espace de nom comme `ns_racine\monrepertoire`, où `ns_racine` désigne de manière générique la racine du système. L'administrateur peut aussi définir ses propres éléments, dans la cadre d'un parc informatique identique, en utilisant par `ns_monrepertoire`.

De manière analogue, nous pouvons faire la même chose pour le registre, illustré par le listing 3.7.

```
NS(HKEY_LOCAL_MACHINE) = ns_hklm
NS(HKEY_USERS) = ns_hku
```

Listing 3.7 – Correspondance entre les ruches du registre système et leur nom dans l'espace de noms

Les sujets Les sujets sont les entités du système qui réalisent les interactions sur les objets. Dans la pratique, les sujets sont un sous-ensemble de l'ensemble des objets. En effet, comme les sujets sont associés aux processus et aux services, ils peuvent interagir entre eux : un processus pouvant par exemple récupérer un *handle* sur un autre processus dans le but d'interagir avec lui.

A la différence des systèmes Linux où l'extension des fichiers n'est pas représentative de son comportement, sur les systèmes Windows, elle permet de spécifier la nature de la ressource. Ainsi, la reconnaissance des processus, et donc des sujets, se fait par l'extension `.exe` qui leur est associée.

Néanmoins, il existe des processus spécifiques qui ne respectent pas ce schéma, comme le processus `System` associé au PID 4. Cependant, ce processus possède le même nom ainsi que le même PID sur chaque système Windows, il est donc facilement reconnaissable.

Ainsi, nous aurons, en prenant par exemple les sujets `explorer.exe` et `System` les noms associés comme le montre le listing 3.8.

Les opérations élémentaires Les opérations élémentaires sont basées sur les ACL classiques présentes sous Windows. Nous obtenons ainsi les droits :

1. r, pour le droit de lecture ;
2. w, pour le droit d'écriture ;
3. x, pour le droit d'exécution.

À ces droits classiques, nous rajoutons les droits suivants :

1. h, pour cacher un élément aux autres entités du système ;
2. ap, pour ajouter des privilèges ;
3. rp, pour supprimer des privilèges.

Les rôles Dans ce modèle de politique, la notion de rôle pour les utilisateurs est très importante. En effet, une politique de type PBAC donne les mêmes droits à tous les utilisateurs, sans distinction aucune. Il est donc nécessaire de définir des rôles pouvant accéder à certains domaines privilégiés pour faire de la séparation de privilèges.

```
grammar pbac;

// include gram_commune

classe :  Chaine ;
autorise_interaction : ;
audit : 'audit ';
autorise_objet : '';
autorise_role : ('role');

contexte :  Chaine ;

operations_elementaires : ' ' operation_elementaire ('\n')?;
operation_elementaire : ('r'|'w'|'x') ;
role:  Chaine ;

 Chaine  : ( ( 'a'..'z'|'A'..'Z'|'0'..'9'|'_'|'.'|','|'\\') )+ ;
Delimiteurs_debut: ' {' (''|'\n' )? ;

Delimiteurs_fin: (' ')? '}' ;
```

Listing 3.9 – Définition des terminaux pour le modèle de protection PBAC

```
systemroot\explorer.exe {
                    systemroot\cmd.exe w
                    systemdrive\fichier.txt r
}
audit explorer.exe { programfiles\firefox.exe r }
role user admin
```

Listing 3.10 – Extrait d'une politique PBAC

Sous Windows, on peut se baser sur les groupes d'appartenance des utilisateurs pour établir les rôles de chaque utilisateur. Le principal avantage de cette méthode est que ce mécanisme est intégré à Windows et facilement modifiable.

Grammaire spécifique pour le modèle PBAC

À partir de la grammaire générique que nous avons définie dans le listing 3.1, nous pouvons étendre cette grammaire pour le modèle PBAC.

Le listing 3.9 détaille la grammaire spécifique pour PBAC. Cette extension ne fait que renseigner les terminaux de la grammaire, qui sont spécifiques à ce modèle de protection.

Cette grammaire spécifique pour le modèle de protection PBAC permet de définir les éléments de la politique : les contextes, les rôles et les opérations élémentaires. Elle va aussi spécifier les délimiteurs pour l'écriture des vecteurs d'accès.

Dans cette politique, les contextes, en pratique des noms symboliques absolus, sont définis comme des chaînes de caractères spécifiques (par exemple incluant les antislash).

Les opérations élémentaires sont représentées par les ACL classiques présentes sur les systèmes : `r` pour l'opération `read`, `w` pour l'opération `write`, etc.

Nous proposons dans le listing 3.10 un extrait de politique pour le modèle PBAC.

Les lignes 1 à 4 autorisent deux interactions directes. La ligne 5 définit une interaction directe qui doit être auditée. Enfin, la ligne 6 permet à un utilisateur du rôle **user** de changer son rôle pour atteindre le rôle **admin**.

3.2.2.2 Domain and Type Enforcement

Représentation du système

Dans le modèle de protection basé sur DTE, les sujets sont associés à des domaines alors que les objets sont associés à des types. Nous allons décrire ici comment construire ces deux notions. Puis, comment proposer des contextes de sécurité étendus avec des notions d'identité et de rôle.

Les objets Les types, qui sont donc associés aux objets du système, sont construits à partir de deux éléments. Le premier élément est simplement le nom relatif de la ressource sans son extension, dans le cas où elle en possède une. Ce nom relatif n'est pas unique sur le système, puisqu'il ne se base que sur le nom de la ressource sans son chemin dans l'arborescence du système de fichiers. Le second élément est un suffixe qui précise la nature de l'objet : répertoire, exécutable, bibliothèque, etc.

Par exemple, la ressource `C:\Program Files\Mozilla\Firefox.exe`. Le nom relatif de la ressource sans son extension est `Firefox`. On ajoute à cela sa caractéristique, ici $Cat(executable)$. Donc le type associé au fichier `C:\Program Files \Mozilla \Firefox.exe` est $Firefox.Cat(executable)$.

De manière similaire, nous aurons des catégories pour les répertoires, les *driver*, les bibliothèques, etc.

Les éléments du registre seront typés de manière similaire au système de fichiers, à la différence qu'ils disposent de catégories spécifiques telles que $Cat(key)$ et $Cat(value)$. Cette dernière catégorie est aussi utilisée pour la prise de décision.

Les sujets Les sujets sont associés à la partie domaine du modèle DTE. À la différence des objets qui possèdent une catégorie, les sujets n'en ont pas besoin. En effet, il n'est pas nécessaire de renseigner la catégorie puisqu'il n'y a que la catégorie processus (en général).

Le domaine est uniquement composé du nom relatif du fichier exécutable privé de son extension. Par exemple, le domaine associé au fichier `C:\Program Files\Mozilla\Firefox.exe` sera `Firefox`. Tout comme pour le modèle PBAC, il existe des cas particuliers tels que le processus nommé `System` qui aura pour domaine exactement le même nom.

Les opérations élémentaires Dans ce modèle de protection, les opérations élémentaires sont dérivées des appels système.

En définissant la fonction Cl, qui associe à un objet o appartenant à l'ensemble O des objets du système l'ensemble des opérations élémentaires spécifiques à cet objet, nous pouvons écrire : $Cl(o) = oe_1, ..., oe_n$, avec $oe_i \in OE$ où OE est l'ensemble de toutes les opérations élémentaires.

Par exemple, le fichier `cmd_exec_t` se définit de la manière suivante : $Cl(cmd_exec_t) = file$ et il possède les opérations élémentaires associées aux fichiers $file${read write execute etc.}.

Nous pouvons citer quelques opérations élémentaires simples comme *read*, *write* ou encore *execute*. Il y a aussi *getattr* lorsqu'un processus demande à récupérer les attributs d'un objet. La liste de ces opérations élémentaires est équivalente au nombre d'appels système disponibles pour l'objet considéré.

Extension aux contextes de sécurité Nous définissons des contextes de sécurité plus complets en ajoutant les notions d'identités mais aussi de rôles aux types et aux domaines. Pour la notion de rôles, nous utiliserons la même représentation que pour le modèle PBAC, à savoir l'utilisation des groupes présents sous Windows. Les identités seront liées au SID de l'utilisateur. Les identités

```
grammar dte

// include gram_commune

classe :  Chaine ;
autorise_interaction : ('allow '|'neverallow ');
audit : 'auditallow ';
autorise_objet : 'transition';
autorise_role : 'role';

contexte : (utilisateur ':')? (role ':')? (attribut);

utilisateur:  Chaine ;
role:  Chaine ;
attribut:  Chaine  ;

operations_elementaires :  ' { ' (operation_elementaire ' ')* operation_elementaire ' }';
operation_elementaire: ('read'|'write'| Chaine ) ;


 Chaine  :  ( 'a'..'z'|'A'..'Z'|'0'..'9'|'_'|'.')+ ;
Delimiteurs_debut: ' ' ;
Delimiteurs_fin: '}' ;
```

Listing 3.12 – Définition des terminaux pour le modèle de protection DTE

et les rôles permettent des contrôles supplémentaires même si ceux-ci sortent du cadre de notre travail.

Ainsi, on obtiendra un contexte de sécurité sous Windows de la forme 3.11.

```
SELinux : user:role:type/domaine
Windows : SID:Groupe Windows:type/domaine
```

Listing 3.11 – Représentation d'un contexte de sécurité sous Windows

Les transitions Il existe deux types de transition : la première est la plus courante, c'est le passage d'un type à un domaine. Cela se caractérise par l'exécution d'un fichier qui est dans la $Cat(executable)$. Pour qu'un processus soit autorisé à transiter vers un domaine destination, il est nécessaire que :

1. le domaine source possède une règle d'accès autorisée à exécuter ce fichier, ainsi que toutes les opérations élémentaires annexes nécessaires à ce type d'opération comme le droit de récupérer des informations sur le fichier (*getattr*) ;
2. le fichier exécutable possède le domaine destination dans la politique de contrôle d'accès.

La seconde transition possible est le changement de rôle. Cette transition n'est pas propre au modèle DTE. Cela permet de réduire considérablement la taille de la politique donnant les accès aux rôles plutôt qu'aux utilisateurs directement.

Grammaire spécifique pour le modèle DTE

Comme pour le modèle PBAC, nous étendons la grammaire commune pour qu'elle puisse fonctionner sur ce modèle de protection. Le listing 3.12 illustre les éléments propres à ce modèle.

Nous définissons deux mots clés pour les règles d'autorisation : *allow* et *neverallow*. Le premier permet d'autoriser de manière explicite l'interaction qui est donnée pour le vecteur d'accès. Le second l'interdit de manière explicite. Nous montrerons dans la partie expérimentation comment nous utilisons ces règles d'interdiction. La règle d'audit est signifiée par le mot clé *audit*.

À la différence du modèle PBAC, où le contexte est une chaine de caractères représentant un chemin dans le système de fichiers, dans le modèle DTE, il est composé de trois identifiants : un identifiant pour spécifier l'utilisateur, un autre pour le rôle et enfin un attribut. Les deux premiers éléments ne sont pas propres au modèle DTE, mais ils facilitent l'administration du système. L'attribut est soit un domaine, pour un sujet, soit un type pour un objet.

Nous proposons une politique minimale dans le listing 3.13 illustrant la grammaire.

```
1 allow explorer_t cmd_exec_t:file { write read getattr execute }
2 auditallow firefox_t document_t:file { read }
3 role user_r sysadm_r
4 transition explorer_t cmd_exec:file cmd_t
```

Listing 3.13 – Extrait d'une politique DTE

La première règle autorise le domaine `firefox_t` à accéder au fichier `cmd_exec_t` en lecture et écriture. La seconde règle est une règle d'audit, mais qui n'autorise pas l'interaction. Enfin, les règles 3 et 4 sont des transitions de rôles et de domaine. Pour que le domaine `cmd_t` soit autorisé, il faut la règle de la ligne 1 du listing ainsi que la ligne 4.

3.2.2.3 Lien entre la politique d'accès direct et le moniteur de référence

La politique d'accès directe (PBAC ou DTE) est parcourue par le moniteur de référence lors de la phase de **décision**. Cette décision peut être divisée en deux : soit le moniteur trouve une règle d'accès dans la politique et il applique le traitement associé, soit il ne trouve rien et dans ce cas-là, il applique une règle par défaut.

La figure 3.28 illustre les deux possibilités offertes par le moniteur. Comme nous pouvons le voir sur la figure, nous avons ajouté une règle à la grammaire `regle_defaut` définissant l'action par défaut que doit avoir le moniteur lorsqu'il ne trouve pas le vecteur d'accès au sein de la politique de sécurité. Dans le cadre d'un moniteur de référence, nous avons choisi d'avoir une règle par défaut qui refuse l'opération et qui génère une trace dans le but de connaître les opérations qui ont été interdites par le moniteur.

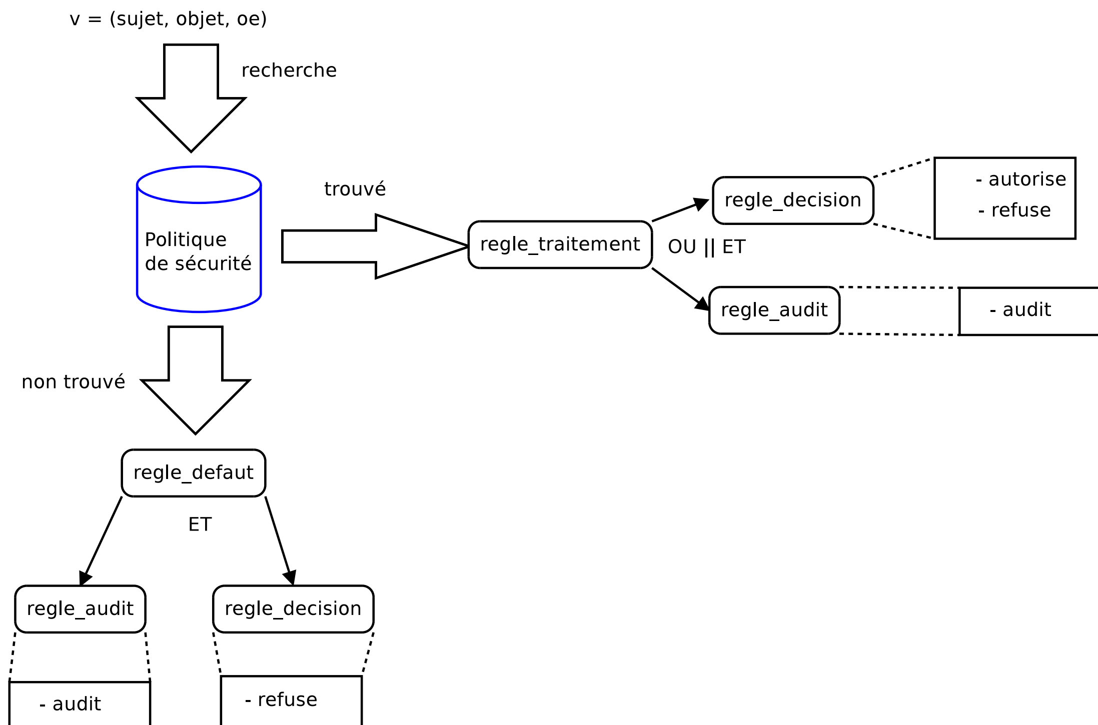

FIGURE 3.28 – Mécanisme de prise de décision du moniteur de référence

3.3 Les techniques de détournement sur les systèmes Windows

Dans cette section, nous allons expliquer les techniques de détournement des flux d'exécution présent sous Windows. Des détails se trouvent en annexe C de notre étude. Nous discuterons des limites de chaque méthode de détournement et de leur place dans le contrôle d'accès sur les systèmes Windows.

3.3.1 Explications des techniques

3.3.1.1 Détournement de la table des appels système

Sur les systèmes d'exploitation Windows, tous les appels système sont stockés dans une table nommée table des appels système. Dans cette table, Windows associe un numéro d'appel système avec une adresse mémoire.

Pour réaliser des interactions, les processus utilisent les fonctions de l'*API Win32*, qui sont les fonctions de haut niveau fournies par Microsoft. Il faut ensuite faire correspondre ces fonctions de haut niveau avec les appels système. Pour cela, une couche d'abstraction placée en espace utilisateur récupère la fonction de l'*API Win32* pour ensuite utiliser l'appel système correspondant.

Pour détourner le flux d'exécution, un *driver* va aller modifier l'adresse d'un appel système, dans la table des appels système, pour la remplacer par l'adresse d'une de ses propres fonctions. Ainsi, lorsque Windows exécute l'appel système dont l'adresse a été modifiée, il exécute une fonction du *driver*.

Le schéma 3.29 montre le flux d'exécution d'une action sous Windows. Lorsqu'un processus veut effectuer une interaction sur le système, il fait appel à une fonction présente dans la table des appels système qui exécute une fonction au sein du noyau.

Le second 3.30 schéma illustre lui le détournement d'un appel système. Lorsque le processus veut effectuer une interaction, avant de passer dans le noyau, son interaction est détournée par l'observateur, qui va autoriser ou refuser cette interaction.

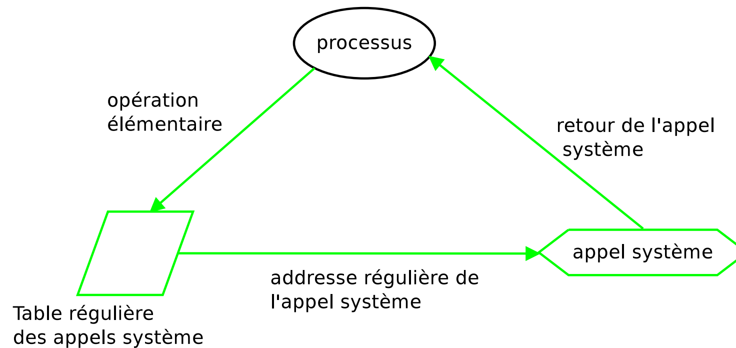

FIGURE 3.29 – Fonctionnement régulier d'un appel système présent dans la table des appels système

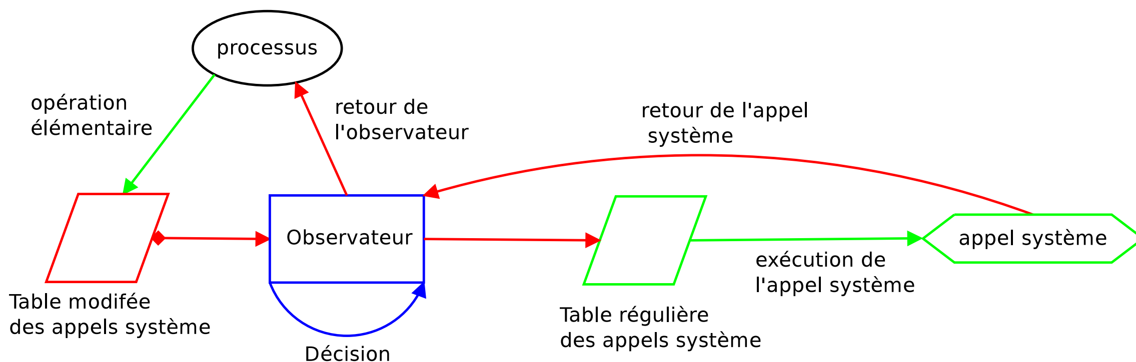

FIGURE 3.30 – Mécanisme de détournement de la table des appels système

3.3.1.2 *Inline-Hook*

Cette technique consiste à modifier directement la fonction en mémoire. Sous Windows, au début des fonctions, il existe une partie de code appelée *junk code* : c'est ce que l'on appelle un *code sans effet de bord*. On peut donc le modifier sans pour autant modifier le comportement de la fonction.

En allant modifier le *junk code* présent au début de la fonction pour y placer un saut inconditionnel, il est possible de détourner le flux d'exécution pour le rediriger vers l'espace mémoire du *driver*.

Cette technique a été implémentée pour les fonctions de l'*API Win32* et est connue sous le nom de *MS Detour* [Hunt et Brubacher, 1999]. Ce projet est porté par *Microsoft Research* et cible essentiellement les systèmes Windows XP. Microsoft propose, au travers d'un *framework*, des API pour modifier de manière complètement transparente des fonctions spécifiques en espace utilisateur grâce à l'utilisation d'un mécanisme nommé *trampoline*. Cette interface est chargée de récupérer les arguments des fonctions pour qu'ils puissent être exploités par la fonction réalisant l'*inline hook*. Elle est cependant transposable aux appels système.

Cette méthode est illustrée par la figure 3.31. Elle détaille le placement, en début de fonction, de la modification introduite pour détourner le flux d'exécution et le placement de l'observateur.

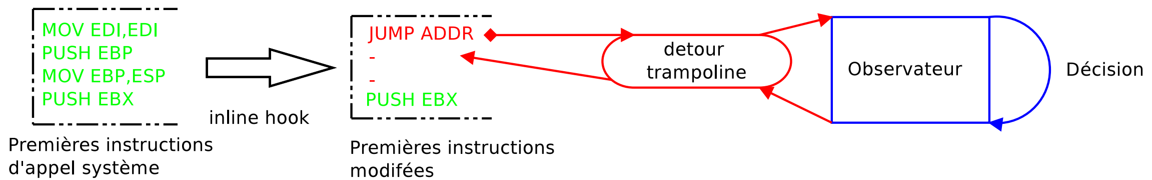

FIGURE 3.31 – Mécanisme d'*inline hook*

3.3.1.3 *filter-driver*

Microsoft a mis en place une architecture spécifique basée sur des *driver* en couches pour contrôler les interactions réalisées sur le système de fichiers. Cette architecture se base sur la création de *filter-driver* s'enregistrant auprès d'un gestionnaire d'entrée/sortie et demandant à traiter certaines IRP (*I/O Request Packets*). Une IRP représente une opération d'entrée/sortie. À la différence des appels système qui ne font qu'une seule opération élémentaire (ouverture de fichier, lecture, écriture, etc.), les IRP englobent plusieurs appels système pour finalement représenter un type d'action, par exemple l'ouverture d'un fichier.

La figure 3.32 montre l'architecture des *driver* en couches. Pour représenter ces différentes couches, qui définissent le comportement du *driver*, les *filter-driver* possèdent une altitude. Cette altitude définit le type du *driver*. Par exemple, il existe une plage d'altitude pour les *driver* d'antivirus ou pour les *driver* de chiffrement. Par construction, les *filter-driver* sont naturellement placé en mode **continuation**.

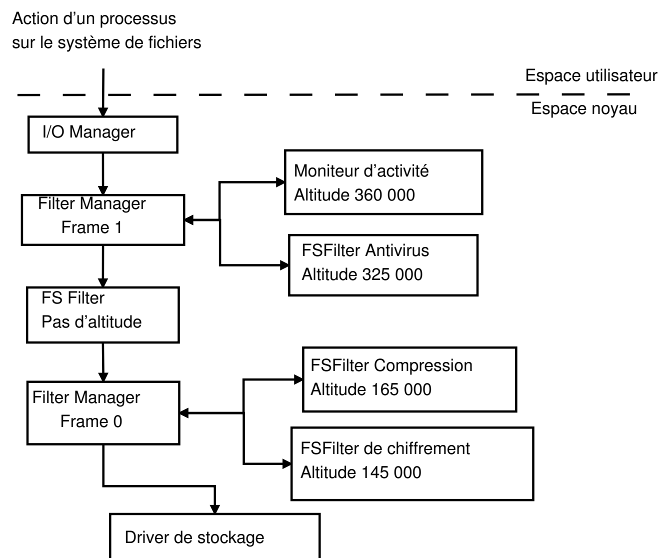

FIGURE 3.32 – Architecture des *filter-driver* en prétraitement

L'altitude du *filter-driver* définit le type du *driver* mais aussi son placement par rapport aux autres *filter-driver*. Ce placement influe sur les informations que reçoit ou non le *driver*.

Une IRP est transmise aux *filter-driver* concernés par l'action courante. Les transmissions aux *filter-driver* sont faites de manière séquentielle et sont définies par leur altitude. Ainsi un *filter-driver* qui est à l'altitude 325 000 récupère l'IRP avant un *filter-driver* qui est à l'altitude 145

000 pour des opérations de prétraitement. A contrario, lors des opérations de post-traitement, le *filter-driver* qui est à l'altitude 145 000 récupère l'IRP avant celui à l'altitude 325 000.

Dans cette architecture, l'observateur se place dans la pile de *driver* comme illustré dans la figure 3.33 pour le prétraitement et pareillement pour le post-traitement 3.34.

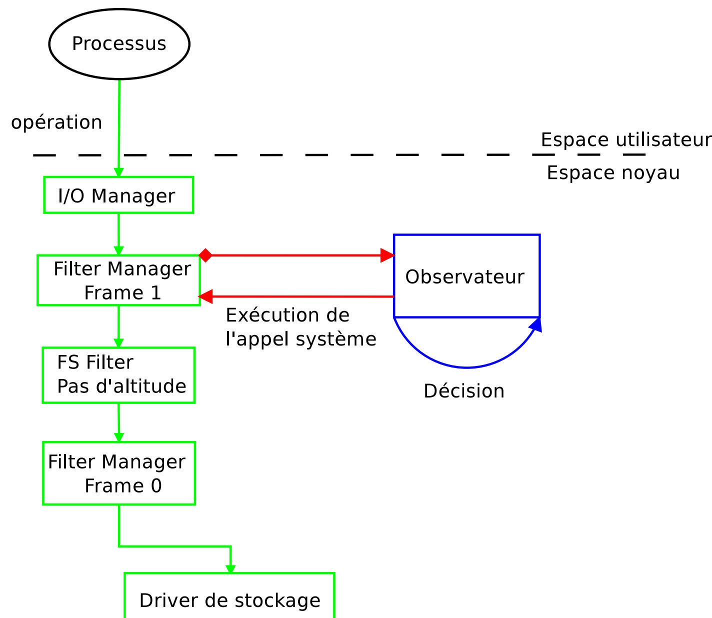

FIGURE 3.33 – Place de l'observateur en prétraitement

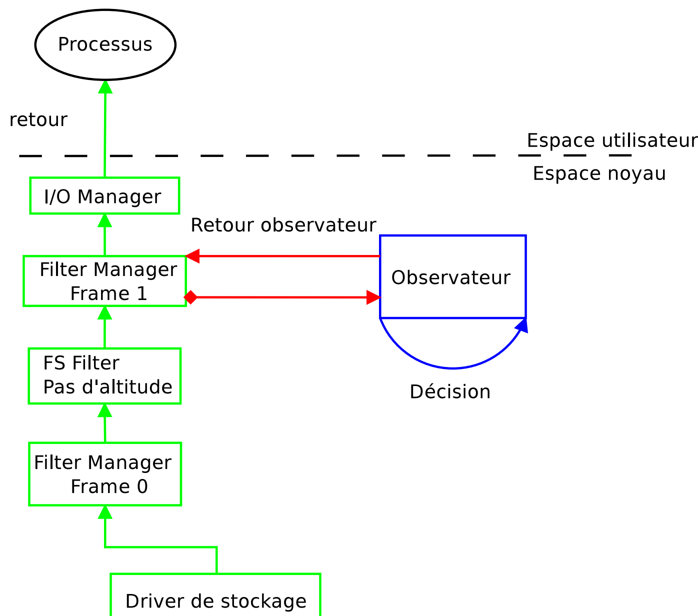

FIGURE 3.34 – Architecture des *filter-driver* en post-traitement

Grâce à cette architecture spécifique, nous pouvons définir à quelle altitude sera notre *filter-driver*. Comme il a pour but de superviser les interactions faites sur le système de fichiers, ce sera un *driver* de type *moniteur d'activité* et aura donc une altitude comprise entre 360000 et 389999. Cette altitude a l'avantage de nous placer très haut dans la pile des *driver*.

3.3.2 Discussion

Nous venons de présenter les méthodes de détournements présentes sous Windows. Nous allons dans cette section discuter du placement de l'observateur dans l'architecture du contrôle d'accès sur les systèmes Windows. Ensuite nous présenterons des méthodes de contrôle des *driver*.

3.3.2.1 Place de l'observateur dans le contrôle d'accès de Windows

Comme nous l'avons déjà présenté dans l'état de l'art (voir la section 2.3.2.1), depuis l'arrivée de Windows Vista, un nouveau mécanisme de sécurité a été introduit dans le système Windows. Il est complémentaire au modèle de protection discrétionnaire classique. Mais à la différence des contrôles d'accès obligatoires présents sous Linux, la vérification du niveau d'intégrité est faite avant la vérification des droits discrétionnaire, ce qui permet par ailleurs, de les contourner.

Avec l'insertion de notre observateur au sein du système Windows, il est nécessaire de connaître sa place dans l'architecture du contrôle d'accès sous Windows. Nous allons ici traiter les trois méthodes présentées précédemment en expliquant le placement de l'observateur pour chaque cas.

Détournement de la table des appels système Lorsque l'observateur détourne le flux d'exécution en modifiant la table des appels système, l'appel système n'est exécuté qu'à partir du moment où l'observateur a autorisé l'interaction.

Cela implique que dans cette configuration, le contrôle s'effectue de la manière suivante :

1. contrôle d'accès obligatoire ;
2. contrôle d'intégrité obligatoire ;
3. contrôle des droits discrétionnaires.

En effet, le détournement empêche l'appel système et donc en pratique les contrôles classiques qui, de ce fait, ont lieu à la suite des contrôles de notre observateur.

Inline hook Cette technique consiste à modifier des fonctions du noyau dans le but de contrôler le plus grand nombre d'opérations. Dans la pratique, si on ne s'intéresse qu'aux appels système, nous aurons donc le même comportement que le détournement de la table des appels système.

Dans cette configuration, le contrôle s'effectue de la manière suivante :

1. contrôle d'accès obligatoire ;
2. contrôle d'intégrité obligatoire ;
3. contrôle des droits discrétionnaires.

filter-driver Dans ce modèle de détournement, l'ordre des vérifications est différent. En effet, dans cette architecture, les requêtes passent par le noyau et sont ensuite transférées au gestionnaire d'entrée/sortie qui les transmet aux *filter-driver* concernés par l'opération.

1. contrôle d'intégrité obligatoire ;
2. contrôle des droits discrétionnaires ;
3. contrôle d'accès obligatoire.

3.3.2.2 Contrôle des *driver*

Toutes ces techniques se basent sur l'utilisation d'un *driver* pour agir en espace noyau et ainsi avoir un impact sur tout le système. Cependant, toutes les opérations que nous avons décrites peuvent être aussi réalisés par n'importe quel *driver*. Il est donc nécessaire de contrôler les autres *driver* qui sont chargés.

Sur les systèmes Windows, les *driver* offrent des moyens de communication entre les périphériques tels que les scanner, les webcam, etc. et le système d'exploitation. Ils sont donc nécessaires au bon fonctionnement du système. On ne peut tout simplement pas empêcher le chargement des *driver*. Il faut par conséquent proposer des solutions pour contrôler aussi les *driver*.

Depuis les systèmes Windows Vista 64 bits et la mise en place de *Kernel Patch Protection*, les *driver* qui sont chargés doivent être signés numériquement par un certificat délivré par Microsoft. Cela constitue une première vérification quant au chargement des *driver*. Cependant, ce mécanisme n'est pas suffisamment fiable puisque certains logiciels malveillants arrivent à charger des *drivers* malgré la présence de ce mécanisme.

Nous allons discuter dans cette section, pour chaque solution, du contrôle du chargement des *driver*. En effet, il est nécessaire les *driver* existant ne puissent pas contourner voire supprimer notre détournement.

Détournement de la table des appels système

Le chargement des *driver* s'effectue par l'appel système `NtLoadDriver`. Il est donc possible, en contrôlant cet appel de contrôler le chargement des *driver*. Cependant, nous allons détailler deux installations de *driver* qui pourraient compromettre le bon fonctionnement de notre moniteur. Le premier exemple concerne un *driver* légitime et le second, un *driver* malveillant.

Un *driver* légitime peut modifier la table des appels système. Ce sont généralement les logiciels de sécurité qui réalisent ce type d'opération comme les antivirus ou les pare-feux.

Si un *driver* légitime modifie la table des appels système, nous pouvons être confrontés à deux possibilités :

1. l'observateur se charge en dernier parmi l'ensemble des *driver* qui modifient la table des appels système : cela implique que notre observateur sera correctement installé et il prendra la première décision.

2. l'observateur ne se charge pas en dernier, c'est-à-dire qu'un *driver*, que l'on a autorisé à se charger, modifie la table des appels système après nos modifications. Si le *driver* fait correctement la modification, c'est-à-dire qu'il prend l'adresse présente dans la table avant sa modification pour appeler l'appel système, alors notre observateur sera toujours impliqué dans la chaine d'exécution. Si le *driver* préfère résoudre manuellement l'appel système, alors l'observateur ne sera plus impliqué dans la chaine d'exécution.

La figure 3.35 illustre l'empilement des *driver* qui détournent des flux d'exécution engendrés par la modification de la tables des appels système par plusieurs *driver*. Chaque observateur maintient une table modifiée des appels système. Le dernier observateur maintient lui la table régulière pour exécuter l'appel système. Dans cette configuration, les observateurs sont en mode **cascade**.

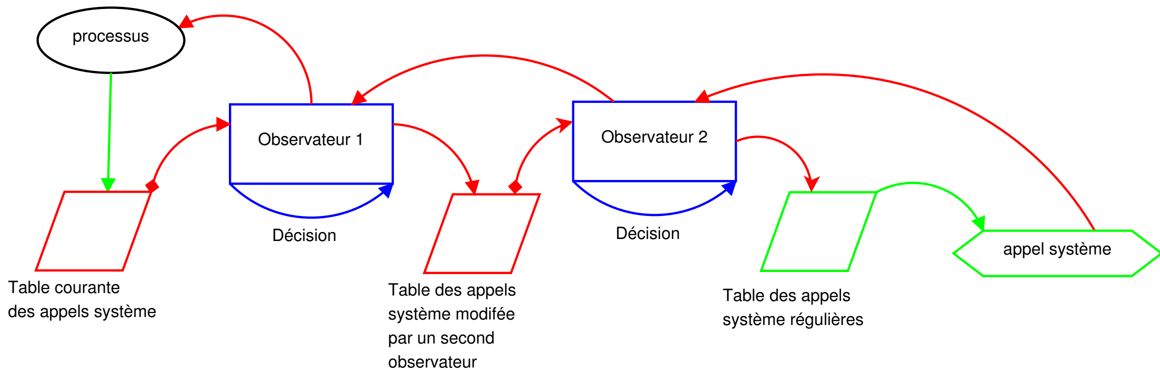

FIGURE 3.35 – Empilement de *driver* modifiant la table des appels système

Un *driver* malveillant peut conduire à une attaque basée sur une situation de concurrence, illustrée par la figure 3.36. Le principe de l'attaque repose sur le fait que les vérifications faites par le *driver* ayant détourné l'appel système ne sont pas atomiques, c'est-à-dire qu'elles prennent un certain temps.

La fonction `NtLoadDriver` prend en paramètre un `UNICODE`. Cet `UNICODE` contient le chemin complet du *driver* à charger. L'attaque réussie lorsque cette fonction est appelée depuis l'espace utilisateur. Au moment du passage en espace noyau, la structure n'est pas recopiée, mais seul un pointeur est copié en espace noyau.

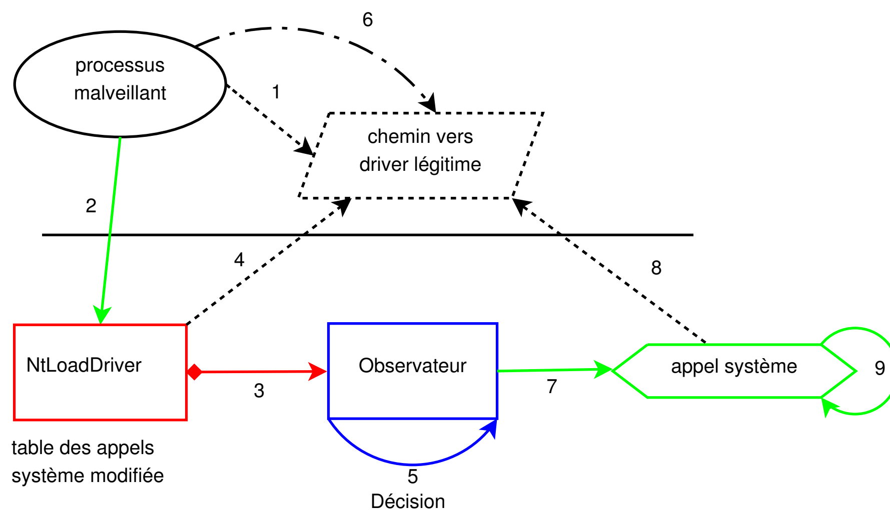

FIGURE 3.36 – Schéma détaillant la situation de concurrence sur la table des appels système

Tout d'abord, un processus malveillant remplit un UNICODE avec un chemin de *driver* légitime (flèche 1), puis exécute l'appel système `NtLoadDriver` (flèche 2). Comme la table des appels système a été modifiée par l'observateur, le flux d'exécution est détourné dans l'observateur (flèche 3). L'UNICODE n'est pas recopié en espace noyau, l'observateur récupère juste un pointeur vers cet UNICODE (flèche 4). L'observateur effectue ses vérifications (flèche 5). Une fois ces vérifications faites, le processus malveillant modifie les informations contenues dans l'UNICODE pour mettre un chemin vers un *driver* malveillant (flèche 6). L'observateur poursuit le flux d'exécution en appelant la véritable fonction (flèche 7). Le noyau récupère lui aussi un pointeur vers cet UNICODE (flèche 8) et exécute la fonction (flèche 9). Il est ainsi possible de charger un *driver* malveillant malgré les vérifications.

Inline hook

Cette technique ne ciblant pas que les appels système, nous pouvons contrôler plus de fonctions impliquées dans le chargement des *driver*. Nous pouvons commencer par contrôler l'appel système `NtLoadDriver`. Nous pouvons aussi cibler `MmLoadSystemImage`. Nous citons ici quelques fonctions mais cette liste n'est pas exhaustive. En effet, beaucoup de fonctions internes ne sont pas documentées donc non présentes sur la *MSDN*.

Tout comme pour le détournement de la table des appels système, si un *driver* effectue la même opération que nous, c'est-à-dire placer un saut inconditionnel en début de fonction, notre observateur ne sera plus impliqué dans la chaîne d'exécution. Il est donc nécessaire de mettre en place un moniteur qui va vérifier que le saut est toujours présent.

A la différence du détournement de la table des appels système, il n'est pas possible d'avoir un *empilement*, comme le montre la figure 3.35 au niveau des *driver* qui réalisent ce genre de détournement. Par conséquent, si un *driver* fait une modification après notre observateur, il faut soit générer une erreur pour bloquer le système, soit remettre en place le saut inconditionnel. Ce n'est pas la seule différence. En effet, ce genre de technique n'est pas une technique couramment utilisée par les *driver* légitimes. Donc, si on est capable de détecter ce genre de modification, on peut légitimement supposer que le *driver* ne devrait pas être autorisé à être chargé. Tout comme pour le mécanisme de détournement de la table des appels système, il est difficile de contrôler une modification qui interviendrait sur notre *driver*. Par conséquent, si un *driver* vient modifier notre observateur pour modifier la fonction responsable de la supervision des modifications, alors l'observateur ne verra pas les nouvelles modifications.

filter-driver

Le *filter-driver* étant la technique préconisée par Microsoft, elle est donc légitimement devenue la méthode utilisée par les logiciels antivirus.

Les *filter-driver* ne contrôlent que les interactions réalisées sur le système de fichiers. Par conséquent, ils ne sont pas capables de contrôler le chargement des exécutables et par extension, des *driver*. C'est pourquoi il est nécessaire de mettre un système de *callback*. Un *callback* peut s'enregistrer auprès du *Process Manager*. Il peut ainsi contrôler le chargement de tous les exécutables, y compris les *driver*. Le contrôle du réseau doit se faire par la création d'un *driver* spécifique, basé sur l'architecture *Windows Filtering Platform*. Ces techniques ne sont pas réservées au *filter-driver*, mais elles sont nécessaires pour contrôler le chargement des exécutables par un *filter-driver*.

Les *filter-driver* sont installés en spécifiant une altitude. Pour respecter les spécifications de Microsoft, il faut faire une demande sur leur site pour se faire attribuer une altitude. Une fois que *filter-driver* possède une altitude fournie par Microsoft, il enregistre cette altitude dans une clé du registre. On peut aussi retrouver toutes les altitudes définies dans un fichier fourni par Microsoft. Néanmoins, rien n'empêchera un *filter-driver* malveillant de se charger à une autre altitude que celle spécifiée dans le registre. Cette définition d'altitude n'implique, en aucun cas, que seul un *filter-driver* précis peut se charger à cette altitude donnée.

3.4 Discussion

Dans ce chapitre, nous avons commencé par formaliser la notion d'observateur. Nous définissons les éléments observés tels que les sujets, objets et opérations élémentaires. Nous avons ensuite défini les modes de sécurité possible pour un observateur : **requête/réponse**, **évidence** et **continuation**. Puis, nous avons détaillé un observateur spécifique : le moniteur de référence. Dans cette

section, nous avons décrit les trois grandes phases de traitement. Ce moniteur de référence sera par la suite, appliqué aux systèmes Windows 7. Enfin, nous formalisons les modes de répartition des observateurs, qui se traduit par des méthodes d'**association**, de **localisation** et de **redondance**.

Nous nous sommes ensuite intéressés à la création de politique d'accès direct. Nous avons, pour cela, proposé un langage générique pour la protection du système définissant une grammaire commune aux modèles obligatoires PBAC et DTE. Pour le modèle PBAC, les politiques sont portables entre les différents systèmes Windows car notre méthode de désignation des ressources est indépendante de la localisation des ressources. Pour le modèle DTE, les ressources étant identifiées par des types ou des domaines, les politiques pourraient être utilisées à la fois sur Linux et sur Windows.

Enfin, nous finissons ce chapitre en décrivant des méthodes de détournement de flux d'exécution sur les systèmes Windows. Au sein d'une discussion ainsi que dans l'annexe C, nous avons décrit quelle est leur place dans le contrôle d'accès sous Windows ainsi que leurs limites. Nous avons expliqué la nécessité de contrôler le chargement des *driver* pour protéger notre observateur.

Il ressort de ce chapitre que nous avons défini de manière générique la notion d'observateur pour les systèmes d'exploitation. Nous avons montré que cette notion pouvait s'appliquer aussi bien aux systèmes Linux qu'aux systèmes Windows. Nous avons abstrait aussi bien leurs modes de sécurité que leurs modes de répartition.

Dans le chapitre 4, nous allons nous intéresser à la répartition de deux observateurs dans un environnement HPC en proposant une architecture originale. Mais nous devrons aussi proposer un modèle nous permettant de mesurer l'impact de la répartition de ces observateurs sur un système.

Dans le chapitre 5, nous mettrons en œuvre le moniteur de référence défini dans ce chapitre pour les systèmes Windows. Pour cela, nous montrerons comment écrire une politique de contrôle d'accès direct et les usages que nous en avons faits pour l'analyse et le contrôle des malware.

Chapitre 4

Répartition d'observateurs en environnement HPC et analyse des performances

En étudiant deux scénarios d'attaque complets sur les systèmes d'exploitation en introduction de notre étude (voir la section 1.3), nous avons expliqué qu'il faillait pouvoir mesurer et contrôler les interactions directes. Ce contrôle s'effectue par la définition d'un observateur capable de gérer les interactions directes. Cependant, il est parfois difficile de prévenir une attaque en agissant uniquement sur les interactions directes sans pour autant nuire au fonctionnement global du système. C'est pour cette raison qu'il est nécessaire d'associer plusieurs observateurs, réalisant des calculs supplémentaires pour protéger le système en profondeur sans nuire à son bon fonctionnement.

Dans le chapitre 3, nous avons proposé une définition générique d'observateur pour les systèmes d'exploitation ainsi qu'un modèle de répartition. Ajouter des observateurs au sein d'un système va nécessairement consommer des ressources supplémentaires, que ce soit du temps processeur, de la mémoire vive, des ressources sur le disque, etc. Cependant, il n'existe pas de modèle de mesure des performances permettant de quantifier de manière précise le surcoût engendré par l'ajout d'observateurs au sein d'un système. Il est pourtant nécessaire de pouvoir calculer ce surcoût, surtout en vue de l'intégration dans des environnements à haute performance.

Nous avons détaillé au cours de l'introduction, mais aussi tout au long de notre étude, qu'il existait des associations *naturelles* d'observateurs sur les systèmes : SELinux et PIGA, SELinux et iptables, un antivirus et un pare-feu sous Windows, etc. Néanmoins, il n'existe pas de modèle permettant de formaliser ces répartitions, c'est pour cela que nous en avons proposé un dans la section 3.1.3. Au cours de cette modélisation, nous avons défini des modes de répartition autorisant le déport de l'observateur.

Dans une première section de ce chapitre, nous allons détailler la méthode que nous avons développée pour mesurer l'impact sur les performances du système d'exploitation de l'association de plusieurs observateurs. Nous traiterons plus spécialement les modes colocalisés et distants avec une association en cascade des observateurs. Puis, dans une autre section, nous détaillerons deux répartitions d'observateurs, SELinux avec PIGA, dans un environnement HPC. La première répartition est colocalisée et la seconde est distante. Nous expliquerons comment nous avons pu déporter PIGA en utilisant les technologies spécifiques des environnements HPC. Enfin nous appliquerons notre méthode de mesure des performances pour calculer l'impact de la répartition des observateurs sur le système. Nous comparerons la répartition colocalisée et notre architecture distante à haute performance. Nous montrerons l'amélioration de la protection grâce aux moniteurs SELinux et PIGA.

4.1 Méthode de mesure des performances d'un système réparti d'observateurs

4.1.1 Mesures globales

L'objectif est l'impact des observateurs répartis sur les performances d'un nœud client. Nous avons choisi de comparer deux observateurs colocalisés et deux observateurs en mode distant. En effet, le mode colocalisé et le mode distant opèrent sur le même principe sauf que le mode distant nécessite des communications réseaux. On pourra ainsi déduire le gain réalisé grâce au mode distant et il est légitime de penser que cela améliorera les performances du nœud client.

Afin d'évaluer l'impact sur les performances du système d'exploitation de ces deux associations, il convient d'avoir un temps de référence (noté $temps_{reference}$). Ce temps de référence correspond au temps pris par un appel système, donc à la différence entre son appel tr_0 et son retour tr_1, sans observateur. Cette méthode de mesure peut être étendue à un jeu de tests, où le premier temps correspond au début du jeu et le second temps à sa fin. Ce temps fait référence au temps sans aucun observateur comme l'illustre la figure 4.1.

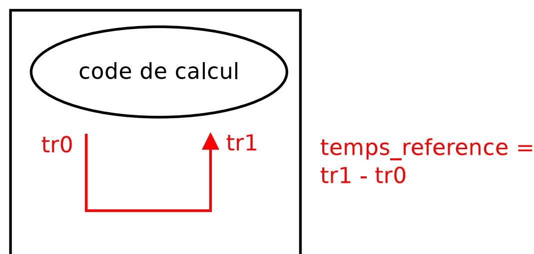

FIGURE 4.1 – Prise du temps de référence

4.1.1.1 Mode colocalisé

Le second temps concerne la latence générée par le premier observateur, noté $temps_{1_coloc_rr} = tl_1 - tl_0$ où tl_1 et tl_0 correspondent respectivement au temps de fin et de début de l'appel système (ou du jeu de test), **colocalisé** en mode **requête/réponse**, illustré par la figure 4.2.

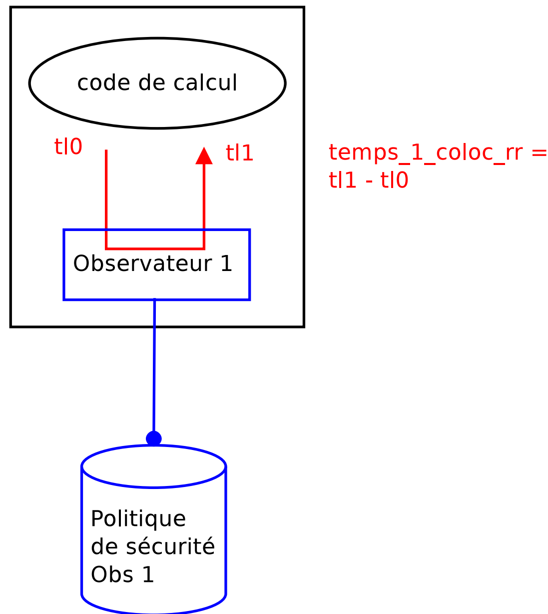

FIGURE 4.2 – Prise du temps avec ajout du premier observateur

Cette mesure, que l'on nomme $temps_{1_coloc_rr}$ comprend :
- le détournement par l'observateur du flux d'exécution ;
- la prise de décision de la part du premier observateur ;
- le retour de l'observateur.

Ces trois éléments mesurés sont illustrés par le diagramme de séquence 4.3.

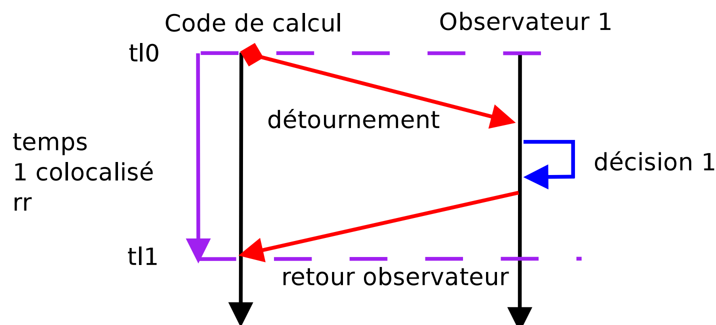

FIGURE 4.3 – Diagramme de séquence illustrant les temps mesurés lors l'ajout d'un seul observateur

Le troisième temps est la mesure du temps global, noté $temps_{2_glob_coloc_rr} = tgl_1 - tgl_0$, avec deux observateurs colocalisés et associés en mode **cascade**. Cette prise de temps est illustrée par la figure 4.4. Dans ce cas, le second observateur est aussi en mode **requête/réponse**.

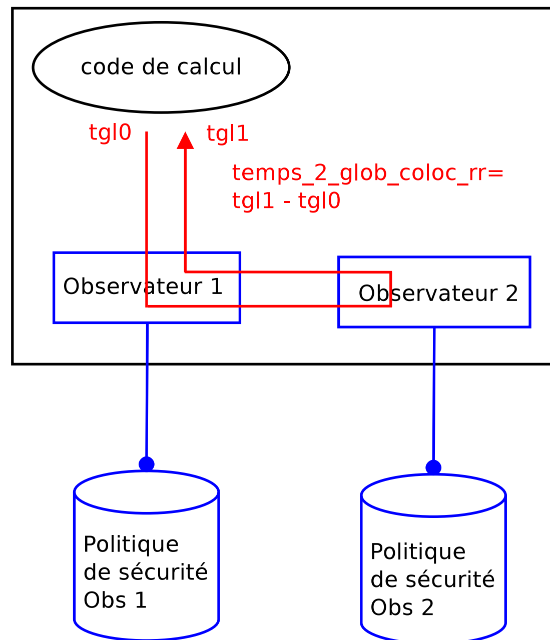

FIGURE 4.4 – Prise du temps global avec deux observateurs en mode colocalisé

Cette mesure, que l'on nomme $temps_{2_glob_coloc_rr}$ comprend :
- le détournement par un observateur du flux d'exécution ;
- la prise de décision de la part du premier observateur ;
- la communication entre les observateurs ;
- le temps de prise de décision du second observateur ;
- la communication retour entre les deux observateurs ;
- le retour du premier observateur.

Ces éléments mesurés sont illustrés par le diagramme de séquence 4.5.

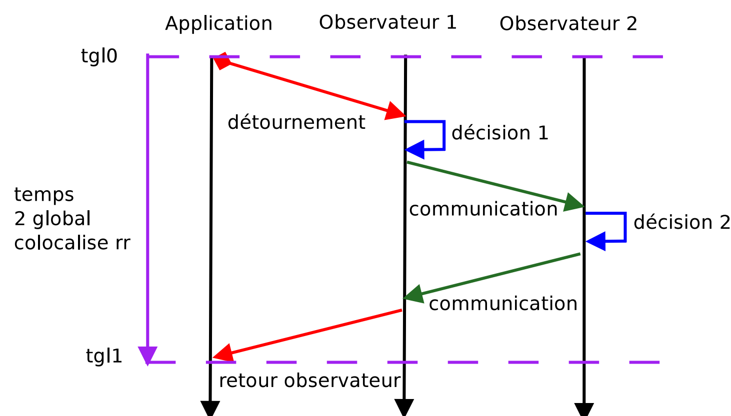

FIGURE 4.5 – Diagramme de séquence illustrant les temps mesurés lors l'ajout de deux observateurs en mode colocalisé

4.1.1.2 Mode distant

Pour la mise en place du second observateur en mode distant, nous introduisons deux nouveaux composants : un **client** et un **serveur**.

Cette quatrième mesure concerne le temps global avec : un observateur local en mode **requête/réponse** associé en mode cascade avec un observateur **distant** en mode **requête/réponse**. Ce temps, noté $temps_{2_glob_dist_rr} = td_1 - td_0$ où td_0 et td_1 correspondent respectivement au début et fin de l'appel système, est illustré par la figure 4.6.

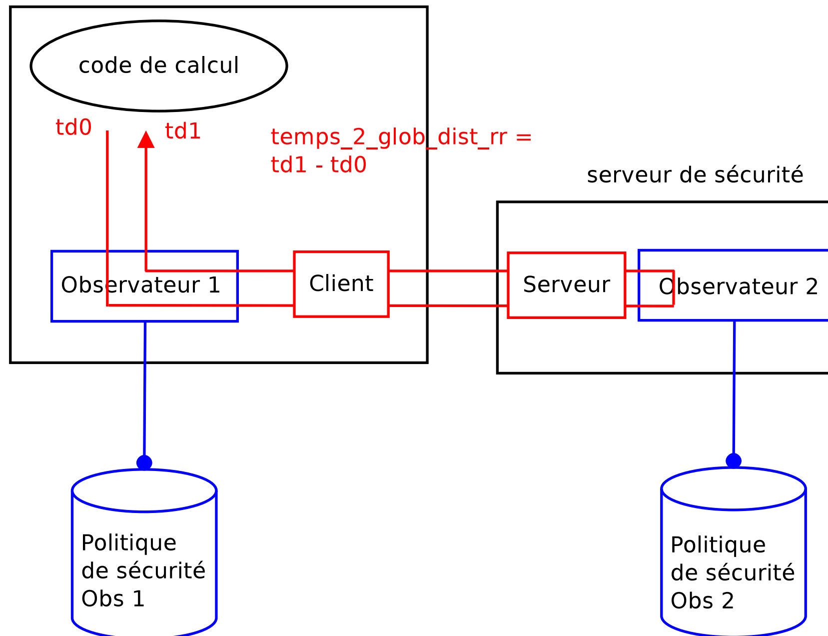

FIGURE 4.6 – Prise du temps global avec un observateur en mode distant

Cette mesure, que l'on nomme $temps_{2_glob_dist_rr}$ comprend :
— le détournement par un observateur du flux d'exécution ;
— la prise de décision de la part du premier observateur ;
— la communication entre le client et le serveur ;
— le temps de prise de décision du second observateur ;
— la communication entre le serveur et le client ;
— le retour du premier observateur.
Ces éléments mesurés sont illustrés par le diagramme de séquence 4.7.

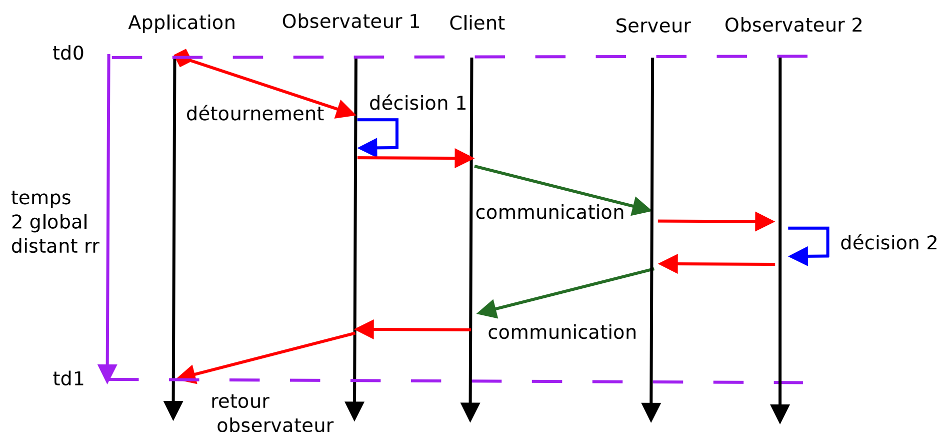

FIGURE 4.7 – Diagramme de séquence illustrant le temps global distant lors du déport du second observateur

4.1.2 Comparaison des performances

4.1.2.1 Combinatoire

En reprenant les modes de répartition que nous avons définis dans le chapitre 3, chaque observateur peut avoir : 3 modes de sécurité, 2 modes d'association et 4 modes de localisation possibles. L'ensemble des configurations possibles est 3x2x4=24 pour un seul observateur. Il semble difficile d'évaluer et comparer toutes ces configurations possibles. Nous avons donc restreint l'étude à certaines configurations pour limiter la combinatoire.

Ainsi, nous considérons les modes **requête/réponse** (protection) et **évidence** (détection) pour deux observateurs en cascade de façon colocalisée et distante.

Le nombre de configurations pour un seul observateur, placé sur la même machine que le client, se résume à 1 localisation, 1 association et 2 modes de sécurité, donc 1x1x2=2. Nous avons donc 2 configurations à évaluer. Pour ce premier observateur, il n'y pas de configuration distante car c'est la solution que nous rencontrons en pratique avec SELinux où le moniteur est en mode colocalisé.

L'ajout d'un second observateur ajoute un certain nombre de configurations. Pour le second observateur, nous avons : 2 localisations, 2 modes de sécurité et 1 association, donc 2x2x1=4. Ce second observateur crée 4 configurations supplémentaires. Nous avons donc 2x4=8 configurations à évaluer, les 2 du premier multipliées par les 4 du second.

Pour résumer, nous devons donc réaliser 8 mesures pour les 8 configurations possibles et les comparer entre elles et avec le temps de référence. A priori, l'instrumentation a peu d'impact et surtout est uniforme avec le temps de référence. En pratique, la réalisation de moyennes sur les temps mesurés lisse la variabilité et le caractère non reproductible des jeux d'essais.

4.1.2.2 Un seul observateur

Le tableau 4.1 établit les mesures à effectuer suivant les modes voulus du premier observateur.

	Mode de sécurité de l'observateur	
Localisation d'un seul observateur	Requête/Réponse	Evidence
Colocalisé	$temps_{1_coloc_rr}$	$temps_{1_coloc_evi}$

TABLE 4.1 – Tableau des mesures à réaliser pour les différents modes d'un seul observateur

Nous retrouvons dans ce tableau les 2 mesures pour un seul observateur placé en mode colocalisé avec le client.

4.1.2.3 Deux observateurs

Le tableau 4.2 établit les mesures à effectuer suivant les modes voulus du second observateur. Chaque mesure doit donc être réalisée suivant les deux modes possibles du premier observateur soit au total deux tableaux 4.2 pour les deux modes de sécurité du premier observateur. Nous ne précisons ici que les temps pour le second observateur.

	Mode de sécurité du second observateur	
Localisation de second observateur	Requête/Réponse	Evidence
Colocalisé	$temps_{2_glob_coloc_rr}$	$temps_{2_glob_coloc_evi}$
Distant	$temps_{2_glob_dist_rr}$	$temps_{2_glob_dist_evi}$

TABLE 4.2 – Tableau des mesures à réaliser pour les différents modes du second observateur

4.1.3 Mesures détaillées

En plus des mesures globales que nous venons de présenter, nous allons spécifier des mesures plus détaillées. Nous n'avons réalisé ces mesures que pour le second observateur en mode **requête/réponse** étant donné que c'est celui-ci qui introduit le plus d'attente pour l'appel système.

La cinquième mesure concerne, puisque nous sommes dans le mode distant, le temps de communication ainsi que le temps de décision du second observateur en mode **requête/réponse** comme illustrée par la figure 4.8. De nouvelles mesures au niveau du client permettent d'évaluer le temps $temps_{comm_observateur2_rr} = tgo_1 - tgo_2$, où tgo_0 et tgo_1 correspondant aux temps d'émission et de réception au niveau du client.

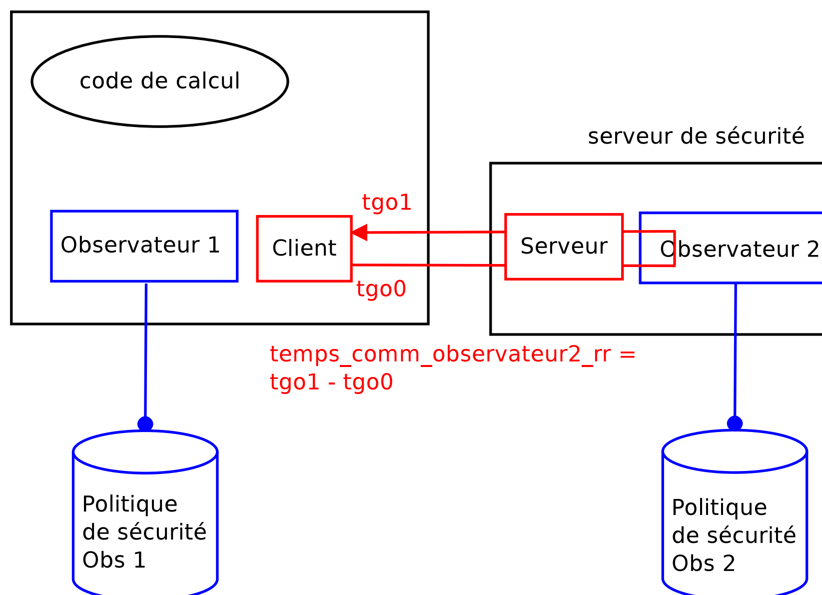

FIGURE 4.8 – Prise du temps avec une communication et un observateur en mode distant

Cette mesure, que l'on nomme $temps_{comm_observateur2_rr}$ comprend :
- la communication entre les observateurs ;
- le temps de prise de décision du second observateur ;
- la communication de retour.

Cette mesure est illustrée par le diagramme de séquence suivant 4.9.

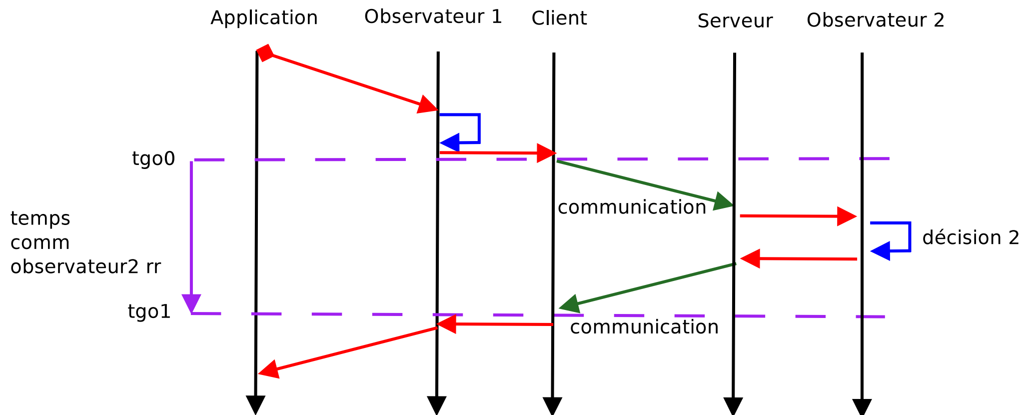

FIGURE 4.9 – Diagramme de séquence illustrant les temps mesurés lors d'une communication complète avec le second observateur en mode distant

Enfin, la sixième mesure concerne le temps de décision du second observateur en mode **requête/réponse**, illustrée par la figure 4.10. Cette mesure se fait au niveau du serveur, on la note $temps_{observateur2_rr} = tod_1 - tod_0$, où tod_1 et tod_0 sont les réponses et les requêtes envoyées depuis le serveur.

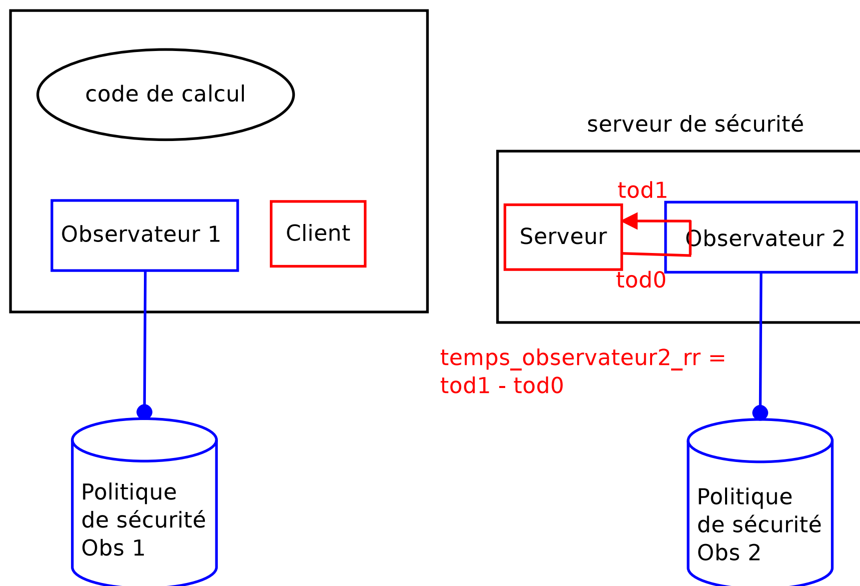

FIGURE 4.10 – Prise du temps pour l'observateur 2 en mode distant

Cette mesure, que l'on nomme $temps_{observateur2_rr}$ comprend :

— Le temps de prise de décision du second observateur.

Cette mesure est illustrée par le diagramme de séquence suivant 4.11.

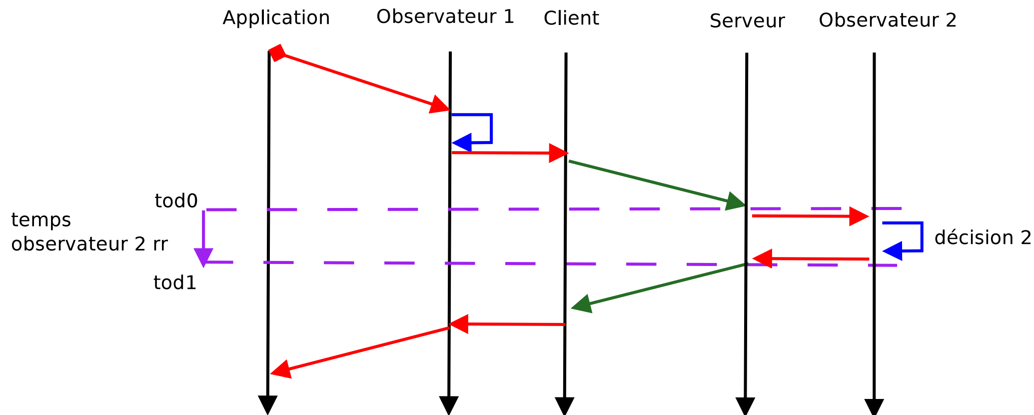

FIGURE 4.11 – Diagramme de séquence illustrant les temps mesurés lors de la prise de décision du second observateur en mode distant

4.1.3.1 Résultats déduits

Nous venons de détailler des mesures globales et des mesures détaillées. Mais certains temps ne sont pas mesurables, comme le temps de communication entre le client et le serveur. Cependant, il est calculable.

Ainsi, nous proposons un ensemble de calcul de performance issu des mesures effectuées :

— **Le temps de communication** : nous n'avons pas défini de mesure pour déterminer le temps pris par la communication entre les deux observateurs. Il est effet difficile de mesurer de manière précise une communication réseau entre deux machines car il faut synchroniser parfaitement les horloges des deux systèmes. Cependant, ce temps est obtenu en soustrayant deux mesures.

$$2 * temps_{communication} = temps_{comm_observateur2_rr} - temps_{observateur2_rr} \quad (4.1)$$

— **Le temps global** : même si nous réalisons une mesure du temps global, nous pouvons la vérifier en additionnant des points de mesure.

$$temps_{2_glob_dist_rr} = temps_{1_coloc_rr} + temps_{communication} * 2 + temps_{observateur2_rr} \quad (4.2)$$

— Nous pouvons en déduire **la latence** introduite par l'ajout des deux observateurs en mode colocalisé, qui s'obtient de manière naturelle en comparant le temps de référence au temps global, qu'il soit mesuré ou calculé.

$$temps_{latence_colocalise_rr} = |temps_{2_glob_coloc_rr} - temps_{reference}|$$

— Nous pouvons en déduire **la latence** introduite par l'ajout des deux observateurs en mode distant, qui s'obtient de manière naturelle en comparant le temps de référence au temps global, qu'il soit mesuré ou calculé.

$$temps_{latence_distant_rr} = |temps_{2_glob_dist_rr} - temps_{reference}|$$

— Nous pouvons ensuite comparer les deux modèles de localisation : colocalisé et distant. Nous pouvons ainsi en déduire **l'impact** d'une architecture vis-à-vis de l'autre.

$$impact_{rr} = \frac{temps_{2_glob_coloc_rr}}{temps_{2_glob_dist_rr}}$$

— **Taux d'augmentation** Comme nous comparons chaque valeur mesurée par rapport à un temps de référence, nous pouvons exprimer le taux d'augmentation introduit au niveau de chaque appel système. Par exemple, le taux d'augmentation pour l'observateur 2 en mode colocalisé est :

$$taux_augmentation = \frac{temps_{2_glob_coloc_rr} - temps_{reference}}{temps_{reference}}$$

4.2 Solution pour répartir les observateurs dans un environnement HPC

Dans cette section, nous allons nous intéresser au développement d'une solution pour répartir les observateurs dans un environnement HPC.

Nous allons donc détailler la mise en place d'une architecture avec deux observateurs pour les systèmes Linux. Le premier observateur est SELinux tandis que le second est PIGA. Cette répartition se fera suivant deux modes de localisation : le mode colocalisé et le mode distant. Dans l'objectif de limiter l'impact sur les performances du nœud client, nous utilisons un réseau InfiniBand pour faire communiquer les deux observateurs entre eux dans le mode distant.

Cette architecture a fait l'objet de deux publications : [Blanc *et al.*, 2013a] et [Blanc *et al.*, 2013b].

Nous nous sommes intéressés dans la publication [Blanc *et al.*, 2011] à la protection des nœuds de connexion, en composant en mode colocalisé SELinux et PIGA.

4.2.1 Architecture avec deux observateurs en mode distant

Notre modèle de protection se base sur l'ajout de deux observateurs. Nous avons ici choisi d'utiliser SELinux et PIGA, qui réalisent des calculs complémentaires. En effet, SELinux gère les interactions directes alors que PIGA est capable de contrôler les scénarios complets. La figure 4.12 illustre l'architecture globale que nous avons mise en place. Le canal de communication que nous avons choisi est un lien InfiniBand. Nous appelons **serveur de sécurité** le nœud qui héberge PIGA.

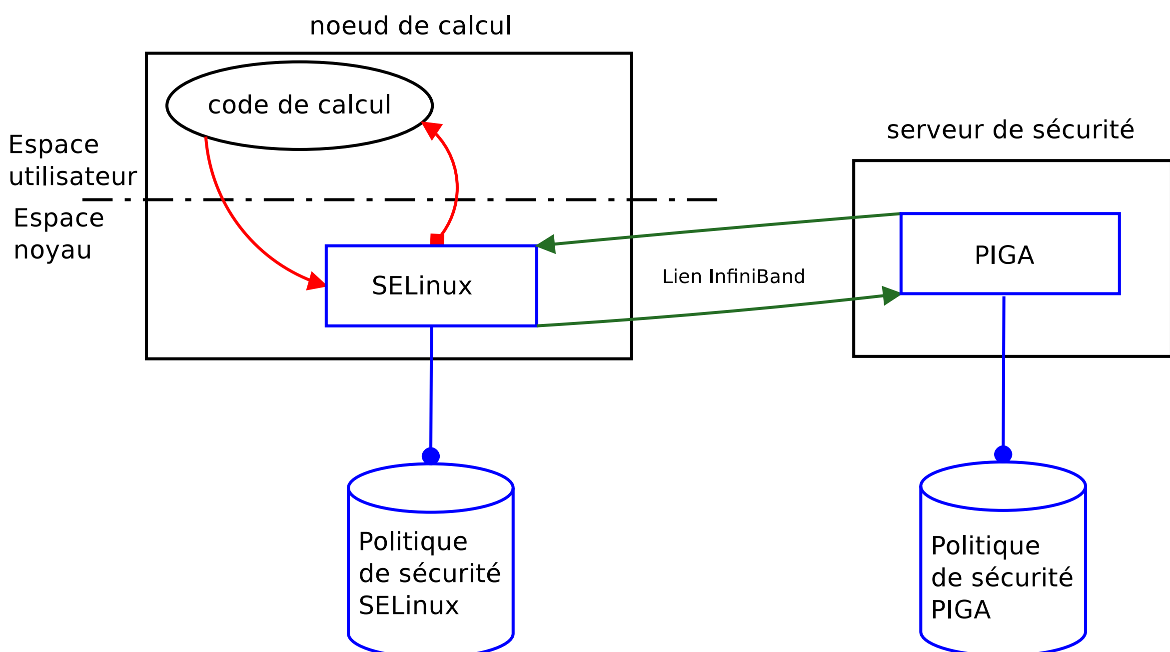

FIGURE 4.12 – Architecture décentralisée

Dans cette architecture, SELinux est chargé de détourner les flux d'exécution du système pour les traiter et les envoyer à PIGA. Il réalise ce détournement grâce à l'utilisation des LSM. Les deux observateurs sont mis en mode de sécurité cascade, ce qui permet d'attendre la réponse de PIGA lorsque celui-ci est en mode protection ou de simplement transférer les requêtes sans attendre de réponse lorsque PIGA est en mode évidence (détection).

4.2.2 Choix des observateurs

4.2.2.1 SELinux

Dans le but de nous rapprocher au maximum des conditions réelles, c'est-à-dire celles mises en place par le CEA, nous avons utilisé un système d'exploitation proche de celui mis en place sur les différents nœuds. Sur les calculateurs tels que *Curie*, c'est le système d'exploitation *Bull Advanced Server* qui est déployé. Cette distribution est un clone du système *Red Hat Enterprise Linux*. Dans ce système d'exploitation, SELinux est intégré et activé par défaut.

SELinux étant nativement intégré au noyau Linux, il n'est pas nécessaire de modifier le code du noyau pour le faire fonctionner. De plus, les politiques SELinux fournies avec cette distribution sont fonctionnelles.

Mais surtout, notre choix a été motivé par les premières expérimentations que nous avons faites sur l'impact de SELinux sur les performances du système d'exploitation. Comme nous le verrons dans la partie résultat de ce chapitre 4.3.2, l'impact de SELinux est minime sur les performances du système. Il peut donc être intégré, même sur les nœuds de calcul. Cela est rendu possible par les différentes optimisations présentes dans SELinux. Nous pouvons par exemple citer le *Access Vector Cache* qui va conserver les derniers vecteurs d'accès calculés par SELinux. Ainsi, si une décision est déjà présente dans ce cache, il n'est pas nécessaire de parcourir de nouveau la politique pour rechercher ce vecteur d'accès. Une seconde optimisation de SELinux est la non revérification des droits acquis lors des opérations de lecture ou d'écriture en l'absence de modification des contextes sujets et objets.

La figure 4.13 illustre le mécanisme de prise de décision de SELinux :

1. un processus réalise une opération élémentaire (un appel système) (flèche 1) ;
2. SELinux la détourne grâce aux LSM (flèche 2) : ces deux premières étapes correspondent à une requête ;
3. il prend une décision au regard de sa politique de contrôle d'accès (flèche 3) ;
4. si l'interaction est interdite, une trace est inscrite dans le fichier d'audit (flèche 4) : cette étape est une réponse de l'observateur ;
5. un code d'erreur est retourné à l'application ayant fait la requête (flèche 5) ;
6. si l'interaction est autorisée, alors SELinux transmet la requête au noyau pour qu'elle soit exécutée (flèche 6) ;
7. le noyau envoie le retour de l'appel système à l'application (flèche 7) : cette étape est une réponse, ici de l'appel système, qui dans ce cas, n'est pas traitée par SELinux.

Chaque décision de SELinux est associée à un *access vector*. Un vecteur d'accès se compose des contextes sujet et objet, de la classe de l'objet, des opérations élémentaires et enfin de la décision d'accès donnée par SELinux.

4.2.2.2 PIGA

Le choix de PIGA a été motivé par le fait qu'il est capable de gérer les scénarios complets en reconstruisant l'activité du système, tout en se basant sur les politiques de SELinux.

PIGA est le second mécanisme de protection utilisé dans notre architecture. Il est constitué de deux parties : PIGA-Kernel, qui détourne les interactions autorisées par SELinux et PIGA-UM, qui est le moteur de décision. Il faut noter que PIGA n'a pas été implanté pour avoir un mode de fonctionnement distant, il nous faut donc résoudre ce problème.

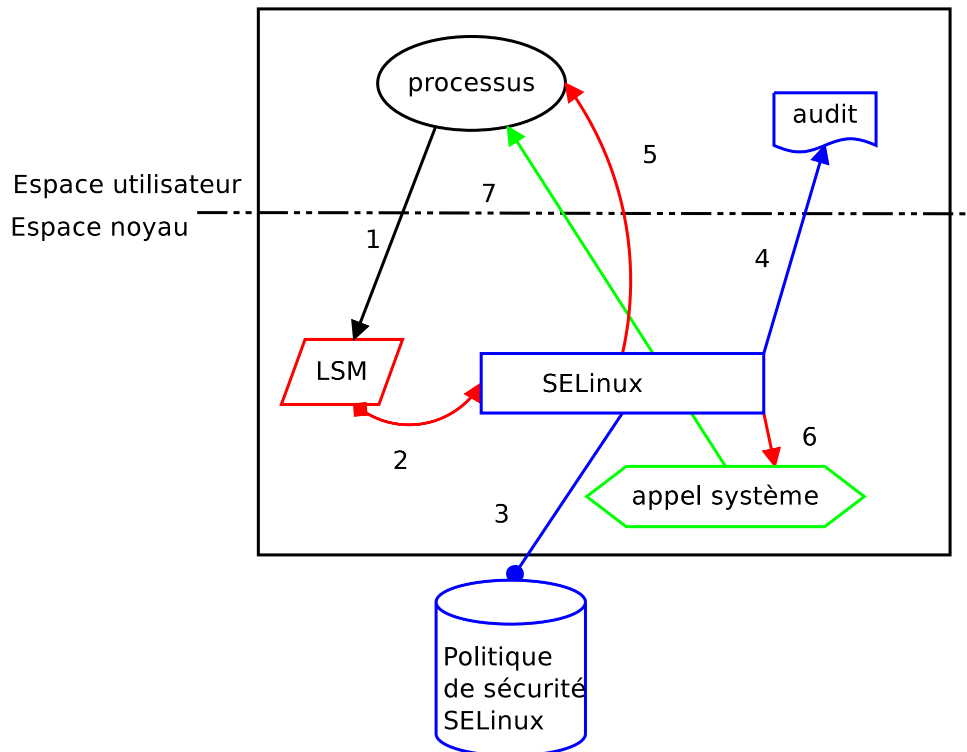

FIGURE 4.13 – Architecture de prise de décision de SELinux

PIGA-Kernel Comme nous l'avons déjà dit, PIGA est basé sur la reconstruction des activités du système. Pour cette reconstruction, PIGA se base sur les vecteurs d'accès générés SELinux. Pour récupérer ces vecteurs, PIGA-Kernel insère un *hook* directement dans le code de SELinux pour détourner le flux d'exécution.

Une fois le flux d'exécution de SELinux détourné, PIGA-Kernel construit sa propre trace avant de l'envoyer à PIGA-UM et il attend une réponse lorsqu'il est placé en mode protection, sinon il rend la main à SELinux. Une fois que la réponse a été calculée, il poursuit l'exécution. Si PIGA-UM n'autorise pas l'interaction, PIGA-Kernel modifie le vecteur d'accès de SELinux afin que l'interaction soit refusée, sinon la décision d'accès prise par SELinux est appliquée.

PIGA-UM PIGA-UM récupère la trace envoyée par PIGA-Kernel pour reconstruire les activités et vérifie que cette interaction ne fait pas partie d'un ensemble d'activités illégitimes.

Une fois que la décision est prise, PIGA-UM la renvoie à PIGA-Kernel. L'insertion des composants PIGA-Kernel et PIGA-UM dans le système, en mode colocalisé, est détaillée dans la figure 4.14.

1. un processus réalise une opération élémentaire (un appel système) qui est détournée par SELinux(flèche 1) : cela correspond à la requête ;
2. SELinux prend une décision au regard de sa politique de contrôle d'accès (flèche 2) ;
3. si l'interaction est autorisée, PIGA-Kernel détourne le flux d'exécution (flèche 3), si l'interaction est refusée par SELinux, le refus est envoyé au processus appelant (flèche 10), cette étape constitue alors une réponse ;
4. PIGA-Kernel envoie la trace à PIGA-UM (flèche 4). En mode détection, il n'attend pas le retour de PIGA-UM et il redonne la main à SELinux (flèche 6) ;

5. PIGA-UM vérifie que cette interaction ne termine pas un scénario complet (flèche 5) ;
6. PIGA-UM envoie une réponse à PIGA-Kernel (flèche 6) ;
7. PIGA-Kernel transmet la réponse à SELinux (flèche 7) ;
8. si l'interaction est refusée, SELinux envoie le refus au processus appelant (flèche 10) : cette étape constitue une réponse ;
9. si l'interaction est autorisée, alors l'appel système est exécuté (flèche 8) ;
10. se noyau envoie le retour de l'appel système à l'application (flèche 9) : cette étape est une réponse, ici de l'appel système, qui dans ce cas, n'est pas traitée par SELinux.

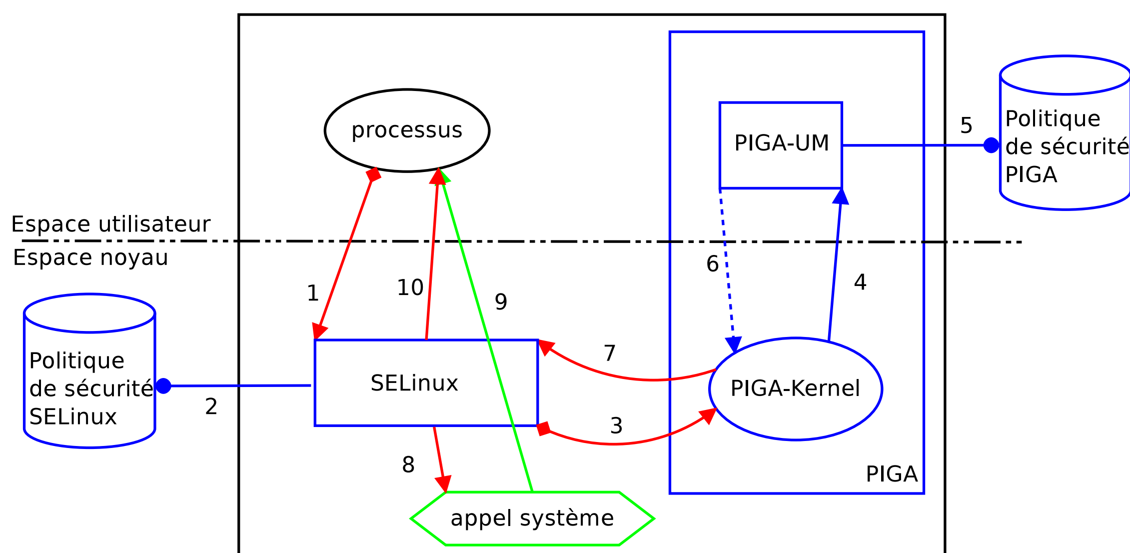

FIGURE 4.14 – Insertion de PIGA dans l'architecture en mode colocalisé

Il est à noter que PIGA-Kernel autorise l'interaction sans attendre de réponse de la part de PIGA-UM lorsque celui-ci est en mode évidence (ou détection). Dans ce mode, on voit que PIGA est moins pénalisant, au niveau de l'impact sur les performances, pour les processus appelant.

4.2.2.3 Infiniband

Pour assurer les communications entre le nœud client et le serveur de sécurité, nous avons fait le choix d'utiliser les réseaux Infiniband. Ce sont des réseaux à très haute performance utilisés dans les environnements HPC qui offrent un important débit et ont une faible latence.

La principale particularité de l'architecture Infiniband est qu'elle peut s'appuyer sur différents protocoles pour être intégrée dans des environnements hétérogènes. Il existe plusieurs types de protocoles : *Remote Direct Memory Access*, *IP over IB*, *ISCSI*, etc. Nous avons fait le choix du protocole RDMA car c'est le protocole qui est le plus efficace lors de l'utilisation d'InfiniBand. Le principal avantage du protocole RDMA est qu'il passe directement par le matériel. Le protocole RDMA peut être utilisé suivant deux méthodes : soit la carte du nœud écrit dans la mémoire distante, soit c'est la carte distante qui vient lire dans la mémoire de la première. Nous avons fait le choix d'écrire dans la mémoire distante.

La figure 4.15 illustre le fonctionnement d'une communication Infiniband sur RDMA. Un processus veut écrire dans la mémoire d'un processus placé sur une autre machine. Le noyau alloue au *driver* de la carte Infiniband un espace mémoire (ou plusieurs). Cet espace mémoire sera le tampon d'échange (lecture ou écriture) pour les deux processus impliqués dans la communication.

Le processus va écrire dans la mémoire allouée (flèche 1) puis valider l'écriture auprès de la carte de la machine courante. Cette dernière va alors récupérer les informations de la mémoire (flèche 3) pour les écrire directement dans la mémoire du processus distant en utilisant le lien InfiniBand (flèche 4). Une fois que la carte a fini son opération d'écriture (flèche 5), l'écrivain notifie le lecteur que l'information est disponible (flèche 6) au travers d'une communication InfiniBand. Le lecteur peut accéder au contenu de la mémoire pour récupérer l'information (flèche 7).

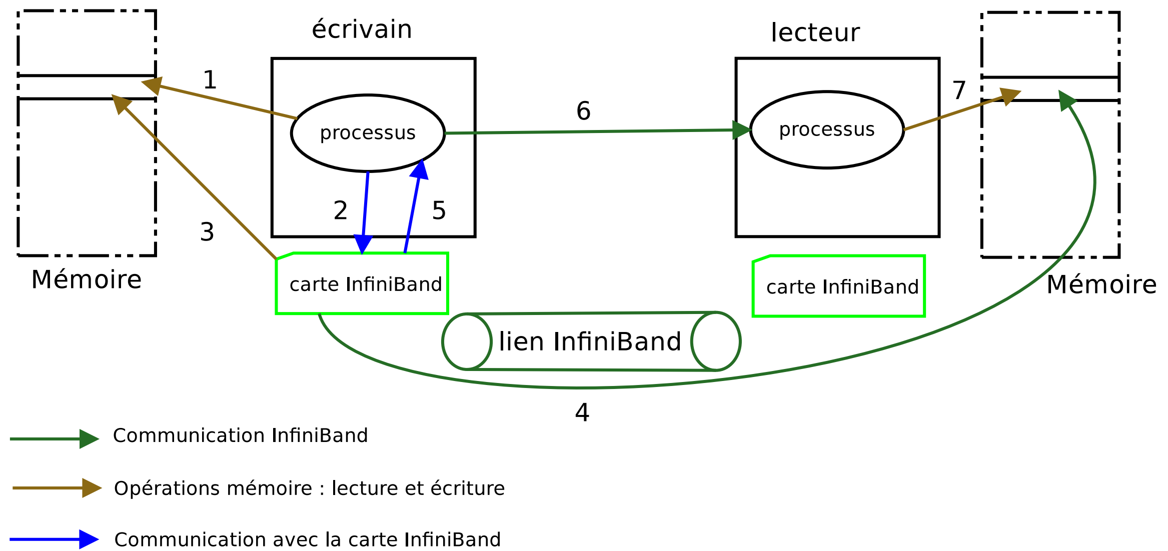

FIGURE 4.15 – Schéma d'une communication RDMA sur un lien InfiniBand

Au cours de cette opération, le noyau n'a jamais été impliqué dans la communication. Ainsi, nous évitons un changement de contexte pour le processus, la création d'un paquet TCP, le parcours des différentes couches du modèle de réseau OSI, etc.

4.2.3 Déport du mécanisme de prise de décision

Nous venons de détailler le fonctionnement en local de SELinux, de PIGA et le fonctionnement du protocole RDMA sur des réseaux InfiniBand. Nous allons maintenant, à partir de tous les éléments proposés, définir une architecture permettant de déporter le mécanisme de protection PIGA.

Nous proposons un programme que l'on nomme **proxy**, chargé des échanges entre PIGA-UM et PIGA-Kernel.

Le proxy

Le proxy est un programme que l'on va retrouver à la fois sur le nœud client, mais aussi sur le serveur de sécurité. Sur le nœud client, il est chargé de récupérer les vecteurs d'accès générés par SELinux et transmis par PIGA-Kernel pour les envoyer sur le lien InfiniBand. Il attend ensuite la réponse provenant du proxy serveur de sécurité pour la transmettre à SELinux. Le proxy serveur récupère les vecteurs d'accès envoyés par le client, les transmet à PIGA-UM, puis renvoie la décision d'accès de PIGA-UM sur le lien InfiniBand.

4.2.3.1 Le nœud client

Nous allons présenter l'architecture simplifiée mise en place sur un nœud client, illustrée par la figure 4.16.

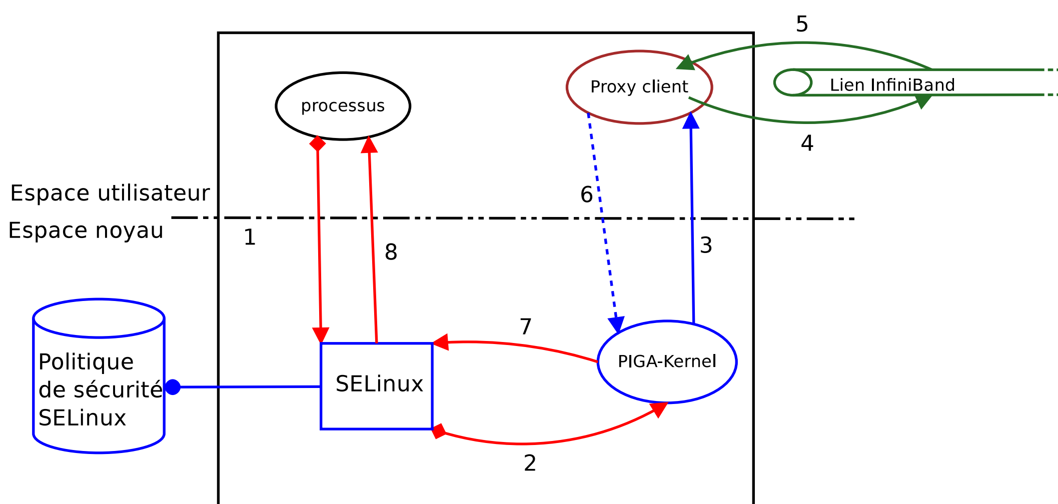

FIGURE 4.16 – Architecture de protection sur un nœud client

Comme nous l'avons décrit auparavant, SELinux détourne le flux d'exécution régulier pour prendre une décision (flèche 1). Si l'interaction est autorisée par SELinux, PIGA-Kernel intercepte à son tour le flux (flèche 2) pour le transmettre au proxy client (flèche 3), qui va l'envoyer au serveur (flèche 5) par le lien InfiniBand.

Le proxy client attend la réponse du proxy serveur et la récupère toujours sur le lien InfiniBand (flèche 5). Il transmet la réponse à PIGA-Kernel (flèche 6) qui la renvoie à SELinux (flèche 7). Si l'accès est autorisé, le vecteur d'accès n'est pas modifié. Dans le cas contraire, une modification du vecteur d'accès a lieu pour interdire l'interaction. Cette décision est enfin transmise au processus (flèche 8). Si la décision est interdite, elle sera enregistrée dans un fichier d'audit.

La figure 4.17 représente l'architecture complète mise en place sur le nœud de calcul et le chemin du contrôle d'une interaction directe.

1. un processus réalise une interaction directe (un appel système) qui est détournée par la couche LSM ;
2. la couche LSM transmette cette requête à SELinux : c'est la requête ;
3. SELinux prend une décision au regard de sa politique de sécurité ;
4. si SELinux autorise l'opération, alors le vecteur d'accès est détourné par PIGA-Kernel ;
5. PIGA-Kernel transmet la trace au proxy ;
6. le proxy envoie cette trace sur le lien Infiniband ;
7. le proxy récupère la réponse du serveur de sécurité ;
8. il la transmet à PIGA-Kernel ;
9. PIGA-Kernel la renvoie à SELinux ;
10. si l'interaction est interdite (soit par SELinux, soit par PIGA), alors la trace est enregistrée dans le fichier d'audit ;
11. si l'interaction est refusée, un retour spécifique est envoyé au processus : c'est une réponse ;
12. sinon, la requête est transmise au noyau, qui l'exécute ;
13. le noyau envoie le retour au processus, non traité par SELinux, c'est une réponse.

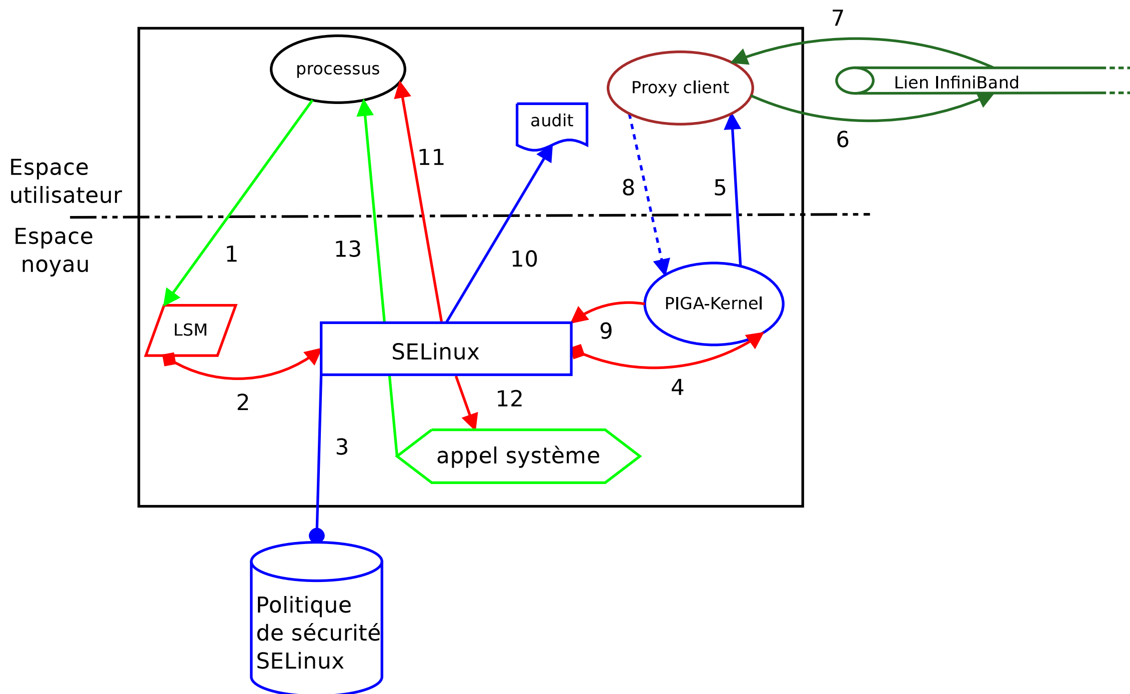

FIGURE 4.17 – Architecture détaillée du contrôle d'accès sur un nœud client

4.2.3.2 Le proxy serveur

Sur le serveur distant, nous n'avons que deux éléments : le proxy serveur et le moniteur de référence PIGA-UM. La figure 4.18 illustre l'architecture du contrôle mise en place sur le serveur de sécurité.

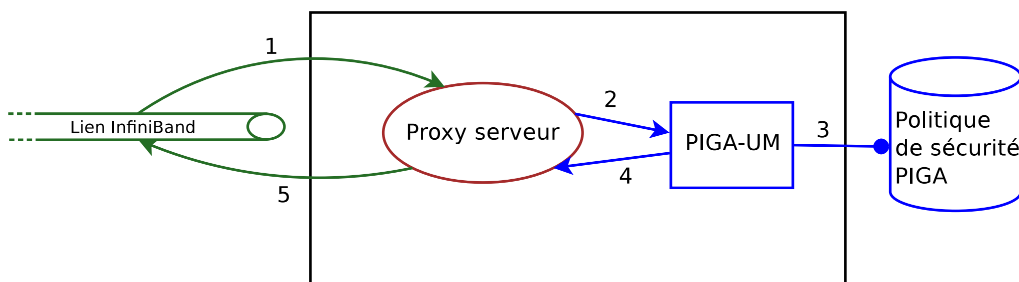

FIGURE 4.18 – Architecture sur le serveur de sécurité

1. Le proxy récupère la trace sur le lien InfiniBand ;
2. Il la transmet à PIGA-UM ;
3. PIGA-UM prend une décision au regard de sa politique de sécurité ;
4. Il envoie sa décision au proxy ;
5. Le proxy la transmet au nœud de calcul en utilisant le lien InfiniBand.

4.3 Expérimentations

Nous venons de détailler la mise en place d'une architecture décentralisée basée sur l'utilisation de deux observateurs : SELinux et PIGA. Pour faire communiquer ces deux observateurs, nous avons implanté un **proxy** qui gère les différentes communications entre les observateurs.

Nous allons maintenant nous attacher à comparer l'impact sur les performances d'une architecture en mode colocalisé avec une architecture où un des observateurs est déporté. Nous allons pour cela, nous appuyer sur le modèle d'analyse que nous avons défini dans la section 4.1.1.

Pour réaliser des mesures de temps précises et surtout reproductibles, nous avons utilisé deux logiciels : le premier se nomme `linpack` et le second est un programme permettant de stresser les observateurs en faisant beaucoup d'entrées/sorties.

4.3.1 Les moyens de mesure

Pour tester l'impact sur les performances de la répartition de deux observateurs, nous avons défini deux types de tests : un test de référence appliqué dans le HPC, et un logiciel spécifique stressant les deux moniteurs de référence SELinux et PIGA.

Le premier test que nous avons réalisé se base sur le logiciel de *benchmark* `linpack` [1]. Ce logiciel est utilisé pour classer les calculateurs dans le monde entier (voir le site Internet `top500` [2]). Nous avons configuré ce logiciel au moyen du fichier de configuration disponible en annexe A. La particularité de ce logiciel est qu'il réalise très peu d'accès disque et d'appels système. C'est essentiellement du calcul en mémoire qui est réalisé, et il va de surcroit utiliser le nombre maximum de cœurs du processeur.

Pour le stress, nous avons créé un logiciel spécifique qui réalise le scénario suivant :

1. ouverture d'un fichier présent sur le système, correspondant à l'appel système `sys_open` ;
2. lecture d'un nombre constant d'octets dans un tampon mémoire, correspondant à l'appel système `sys_read` ;
3. fermeture du fichier, correspondant à l'appel système `sys_close` ;
4. création d'un nouveau fichier avec un nom aléatoire sur le disque local, correspondant à l'appel système `sys_open` ;
5. écriture du tampon dans le fichier, correspondant à l'appel système `sys_write` ;
6. fermeture du fichier, correspondant à l'appel système `sys_close` ;

Le code utilisé est mis en annexe B. Nous avons réalisé ce test pendant une durée d'une heure et les prises de temps sont réalisées en début et fin de séquence pour avoir une vision macroscopique des latences qui sont dues au modèle de protection.

Ce test nous permet d'estimer l'impact du modèle de protection sur le système dans, ce que l'on peut considérer comme le pire des cas pour les mécanismes de protection. En effet, le scénario effectue beaucoup d'appels système puisqu'il ne fait que des entrées/sorties. Toutefois, réaliser des tests sur un système de fichiers peut soulever quelques problèmes. En effet, il ne faut pas que les mesures soient influencées par le **cache** du système de fichiers, ou par la vitesse d'écriture du disque dur. C'est pourquoi, nous avons réalisé la prise de temps en début et en fin de scénario, plutôt qu'entre chaque appel système car cela permet de linéariser les effets des différents caches présents sur le système. Ce test est réalisé de manière séquentielle, c'est-à-dire qu'un seul cœur du processeur est utilisé durant l'expérimentation.

1. `http://www.netlib.org/benchmark/hpl/`
2. `http://top500.org`

4.3.2 Résultats sur les performances

Nous présentons ici les résultats pour les 8 temps globaux que nous avons présentés dans la section 4.1.1. Comme nous l'avons dit, nous allons comparer ces mesures avec une mesure de référence prise sans aucun observateur.

Nous présentons ici les résultats que nous avons obtenus pour comparer les modèles de répartition sur les performances des systèmes. Nous allons découper ces résultats en partant d'une vision globale puis isoler chaque élément de la prise de décision grâce au modèle vu en section 4.1.1.

4.3.2.1 Architecture d'expérimentation

Les expérimentations que nous présentons ici ont toutes été réalisées sur des machines physiques pour être les plus précises possibles. En effet, la virtualisation ajoute une couche de complexité et entraîne des effets de bords qu'il est difficile d'annuler.

Les nœuds clients Nous disposions de deux machines parfaitement identiques pour réaliser des calculs. Nous avons ainsi pu réaliser des tests en mode colocalisé et en utilisant notre architecture décentralisée.

Les nœuds de calcul disposent de deux processeurs *Intel(R) Xeon(R) X5450* cadencés à 3,00 GHz. Ils disposent chacun de 8 Go de mémoire vive. Les tests sont réalisés sur un système de fichiers en `ext4`, avec un matériel en RAID-10, sur des disques en SAS à 15 000 tours par minutes.

Nous avons choisi comme distribution `Scientific Linux 6`, qui un clone de `RHEL 6`. Les paquets formant le système de base sont les mêmes entre les deux distributions. Comme nous avons eu besoin de modifier le noyau, nous avons modifié un `RPM` source de `Red Hat` nommé `2.6.32-279.5.2.el6`.

SELinux a été modifié pour ne charger que 37 modules. Nous avons supprimé du chargement de la politique les modules inutiles sur un nœud de calcul tels que le module `mozilla`, ou encore `openoffice`. La politique SELinux ne contient plus que 79196 règles d'autorisation alors que la politique de référence en contenait 240669.

De plus, nous avons défini des politiques SELinux spécifiques pour que `linpack` et notre logiciel stressant le système de fichiers soient autorisés.

Le serveur de sécurité Le serveur de sécurité est une machine dédiée qui dispose de deux processeurs *Intel(R) Xeon(R) E5405* cadencés à 2,00 GHz. Ce nœud dispose lui aussi de 8 Go de mémoire vive. Le système installé est aussi du `Scientific Linux 6`.

Sur cette machine, nous avons déployé PIGA-UM, qui s'exécute en espace utilisateur, avec l'ensemble de propriété de sécurité du listing 4.1. Ces propriétés de sécurité sont générales et assurent la confidentialité des utilisateurs vis-à-vis des personnes présentes dans le rôle `staff_r` ainsi que des personnes présentes dans le rôle `sysadm_r`, l'intégrité des fichiers de configuration et des domaines `linpack` et `staff_t`. En traitant à la fois de la confidentialité et de l'intégrité, nous configurons PIGA dans le pire des cas car cela génère un grand nombre d'activités à surveiller.

Le canal de communication Les deux machines sont connectées en Ethernet et en InfiniBand. Le nœud de calcul et le serveur de sécurité disposent de la même technologie de carte *Mellanox Technologies MT26428* offrant un débit théorique maximum de 40 Gb par secondes.

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

confidentiality( $sc1:=system_u:object_r:nfs_t, $sc2:=user_u:user_r:user_t );
confidentiality( $sc2:=system_u:object_r:nfs_t, $sc1:=user_u:user_r:user_t );

confidentiality( $sc1:=system_u:object_r:nfs_t, $sc2:=staff_u:staff_r:staff_t )
    ;
confidentiality( $sc2:=system_u:object_r:nfs_t, $sc1:=staff_u:staff_r:staff_t )
    ;

confidentiality( $sc1:="user_u:user_r:user.*_t", $sc2:="staff_u:staff_r:staff.*
    _t" );
confidentiality( $sc2:="user_u:user_r:user.*_t", $sc1:="staff_u:staff_r:staff.*
    _t" );

confidentiality( $sc1:="user_u:user_r:user.*_t", $sc2:="sysadm_u:sysadm_r:
    sysadm.*_t" );
confidentiality( $sc2:="user_u:user_r:user.*_t", $sc1:="sysadm_u:sysadm_r:
    sysadm.*_t" );

integrity( $sc1:="user_u:user_r:user.*_t", $sc2:=".*:.*:.*etc_t" );
integrity( $sc1:="staff_u:staff_r:staff.*_t", $sc2:=".*:.*:.*etc_t" );

domainintegrity( $CHROOT:="staff_u:staff_r:linpack.*" );
domainintegrity( $CHROOT:="staff_u:staff_r:staff.*" );
```

Listing 4.1 – Propriétés de sécurité appliquées pour les tests de performances

4.3.2.2 Mesures globales

L'objectif est tout d'abord de déterminer la latence *globale* au niveau de l'application due à notre modèle de protection.

`Linpack` Ce test a été fait pendant une durée de 23 heures. `Linpack` réalise plusieurs types d'opérations, c'est-à-dire des calculs matriciels, et propose le temps mis pour réaliser le calcul comme le montre l'extrait 4.2. Il réalise 18 opérations différentes. Nous faisons une moyenne du temps mis et de la puissance calculée pour les différents tests que nous avons réalisés.

```
T/V              N    NB    P     Q               Time             Gflops
WR00L2L2     26752  128    2     4            8117.16          1.573e+00
```

Listing 4.2 – Extrait d'un fichier de résultat de `linpack`

Le tableau 4.3 propose les résultats d'un ensemble de tests effectués pour connaître l'impact sur les performances du système d'exploitation des différentes répartitions d'observateurs.

Comme les premiers résultats obtenus avec SELinux (en mode requête/réponse mais aussi en mode évidence) sont concluants, nous avons réalisé les expérimentations suivantes avec SELinux uniquement en mode requête/réponse. Nous comparons donc 6 temps globaux avec le temps de référence. De plus, dans le but de minimiser les traitements réalisés par PIGA, il est nécessaire de bloquer les interactions directes lorsqu'elles ne sont pas autorisées dans la politique de contrôle d'accès. C'est pour cela que nous avons réalisé les tests en ajoutant PIGA uniquement avec SELinux en mode protection.

Pour cela, nous avons tout d'abord réalisé un premier test avec aucune protection obligatoire, que ce soit SELinux ou PIGA. C'est le test de référence. Puis nous avons ajouté un à un chaque

moniteur dans le but d'avoir des résultats précis et exploitables. Nous avons donc ajouté SELinux à nos tests en utilisant les modes : détection et protection. Lors du test de ces deux modes, SELinux disposaient de la même politique de sécurité. On peut noter une légère amélioration des résultats lorsque SELinux est activé sur la machine. Cela peut s'expliquer par le fait que SELinux va bloquer des programmes concurrents et ainsi, `linpack` ne sera pas préempté. Nous avons ensuite ajouté PIGA en mode détection pour avoir une première vision de l'impact de ce mécanisme. Nous avons réalisé ces tests avec une version en mode colocalisé puis avec une version décentralisée. L'objectif étant de savoir si le déport d'un observateur apportait réellement un plus au niveau de l'impact sur les performances du système d'exploitation.

Les résultats obtenus avec `Linpack` montrent que la version décentralisée n'impacte pas les performances du système d'exploitation. La version en mode colocalisé de PIGA génère une faible latence.

	Temps en secondes	Puissance en GFlops
$temps_{reference}$	8 124	1.571
SELinux détection : $temps_{1_coloc_evi}$	8 118	1.572
SELinux protection : $temps_{1_coloc_rr}$	8 111	1.575
SELinux/PIGA colocalisé : $temps_{2_glob_coloc_evi}$	8 127	1.571
SELinux/PIGA colocalisé : $temps_{2_glob_coloc_rr}$	8 130	1.570
SELinux/PIGA distant : $temps_{2_glob_dist_evi}$	8 121	1.572
SELinux/PIGA distant : $temps_{2_glob_dist_rr}$	8 114	1.573

TABLE 4.3 – Tableau des résultats avec l'utilisation de `linpack` avec PIGA

Le tableau 4.3 présente aussi les résultats obtenus, toujours avec `linpack`, mais cette fois-ci, PIGA est mis en mode protection. Dans cette configuration, SELinux est toujours placé en mode *enforcing*.

L'utilisation de PIGA en mode distant améliore légèrement les performances vis-à-vis du mode colocalisé.

Stress des moniteurs répartis Notre second test vise à réaliser un maximum d'appels système afin de solliciter de manière soutenue les observateurs répartis.

Nous rappelons brièvement le protocole de test. Le programme est lancé pendant une heure et réalise en boucle le scénario des actions : ouverture, lecture, fermeture, création, écriture, fermeture. Nous récupérons le nombre de jeux/scénarios réalisés. Le temps indiqué est le temps moyen de chaque scénario correspondant à un jeu de test.

Le tableau 4.4 présente les résultats que nous avons obtenus en stressant le système de fichiers. Dans ce tableau, nous détaillons les résultats obtenus avec le moniteur PIGA mis en mode détection. Dans chaque test, nous avons mis SELinux en mode protection puisque la première mesure, $temps_{1_coloc_rr}$ présente dans ce tableau, confirme que l'impact de SELinux en mode requête/réponse est négligeable par rapport au temps de référence, mais aussi à son mode évidence.

En mode détection, avec SELinux et PIGA en mode colocalisé, nous multiplions par 6 le temps moyen par scénario, ce qui conduit à réduire de moitié le nombre de scénarios. Avec PIGA en distant, nous réduisons cette latence. Ainsi nous obtenons un temps moyen multiplié par 3 et un nombre d'opérations réduit d'un tiers par rapport au mode avec le seul moniteur SELinux.

Le tableau 4.4 donne aussi les résultats obtenus avec PIGA en mode protection. Avec PIGA en mode colocalisé, nous divisons par 5 le nombre de scénarios et nous multiplions le temps moyen par 13. Cependant, nous améliorons avec PIGA en mode distant puisque le nombre de scénarios réalisés est divisé par 2 et le temps moyen est multiplié par 5.

	Nombre de scénarios	temps par scénario en μs
$temps_{reference}$	5 176 731	689
SELinux : $temps_{1_coloc_evi}$	5 176 731	691
SELinux : $temps_{1_coloc_rr}$	5 176 731	692
SELinux + PIGA colocalisé $temps_{1_glob_coloc_evi}$	2 481 727	3 787
SELinux + PIGA colocalisé $temps_{1_glob_coloc_rr}$	1 133 107	7 954
SELinux + PIGA distant $temps_{2_glob_dist_evi}$	3 889 542	1 717
SELinux + PIGA distant $temps_{2_glob_dist_rr}$	2 746 805	3 186

TABLE 4.4 – Table détaillant l'impact avec stress des moniteurs répartis

À partir de ces différentes mesures, obtenues en mode détection et en mode protection, nous pouvons estimer le temps moyen par appel système. Pour cela, nous divisons le temps moyen par scénario par le nombre d'opérations élémentaires réalisées par notre programme, c'est-à-dire 6.

Ces résultats, illustrés par le tableau 4.5, nous permettent, tout d'abord, de calculer un temps moyen par appel système (ou plus exactement par interaction) au niveau de notre système.

Ensuite, nous sommes capables d'estimer le surcoût induit par la répartition de deux observateurs, que ce soit en mode cololocalisé ou en mode distant :

- colocalisé : en mode protection, le surcoût est de 1211 μs par appel système alors qu'en mode détection il n'est que de 517 μs ;
- distant : en mode protection le surcoût est de 417 μs par appel système alors qu'en mode détection, il n'est que de 172 μs.

	temps moyen/scénario en μs	temps moyen/appel système en μs
Sans SELinux : $temps_{reference}$	689	114,3
SELinux $temps_{1_coloc_rr}$	692	115,3
SELinux + PIGA colocalisé : $temps_{2_glob_coloc_evi}$	3 787	631,12
SELinux + PIGA colocalisé : $temps_{2_glob_coloc_rr}$	7 954	1325,6
SELinux + PIGA distant : $temp_{2_glob_dist_evi}$	1 717	286,16
SELinux + PIGA distant : $temp_{2_glob_dist_rr}$	3 186	531

TABLE 4.5 – Table détaillant le temps moyen par appel système

Il faut rappeler que ces surcoûts correspondent à notre avis au pire des cas et que celui-ci est peu représentatif des environnements HPC. En effet, dans la pratique, les codes de calcul ne font pas autant d'entrée/sortie, mais réalisent des calculs en mémoire.

4.3.2.3 Mesures détaillées

Afin de définir où sont les surcoûts des observateurs répartis, nous appliquons notre méthode de mesure définie en section 4.1.1. Dans cette partie, nous nous sommes exclusivement intéressés au mode distant car c'est cette architecture qui vise à être implantée dans un environnement HPC. L'objectif est de donc d'utiliser notre solution pour analyser les performances sur cette architecture. Nous réalisons ces tests uniquement pour le mode requête/réponse de PIGA.

Grâce à nos premiers tests, nous avons montré que ce n'était pas SELinux qui générait une latence. Toutes les mesures vont nous permettre de quantifier les surcoûts aux différents endroits.

Estimation du coût d'une communication complète avec prise de décision du second observateur Nous avons réalisé des mesures de temps au niveau du proxy client déployé sur le nœud client. Cette mesure, $temps_{comm_observateur2_rr}$, consiste à connaitre le temps pris par la suite d'instructions suivantes :

1. le proxy a déjà lu la trace générée par PIGA-Kernel donc cette interaction n'est pas prise en compte dans la mesure ;
2. il prend le premier temps ;
3. il envoie la trace au serveur de sécurité et attend la réponse ;
4. il obtient une réponse ;
5. il prend le deuxième temps.

Le diagramme de séquence 4.19 détaille toutes les opérations qui se déroulent pendant la mesure.

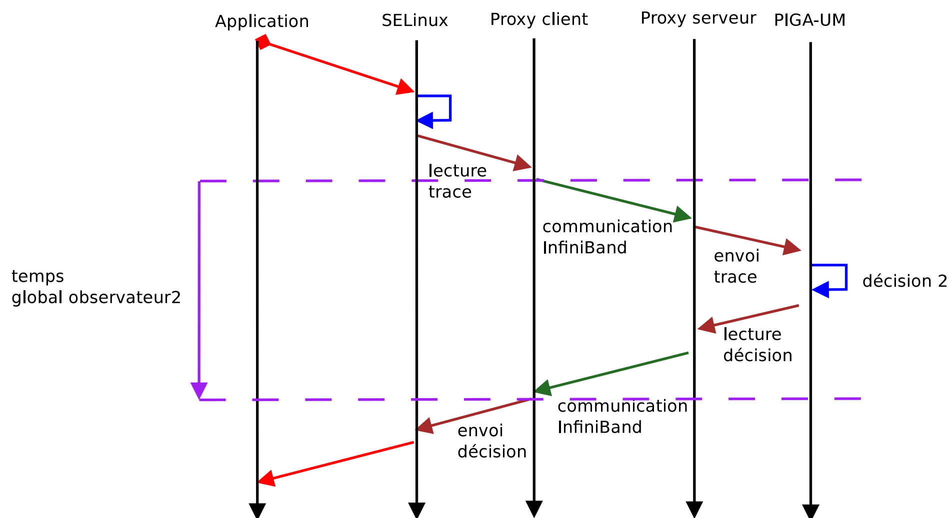

FIGURE 4.19 – Diagramme de séquence illustrant la mesure faite au niveau du proxy coté nœud de calcul

Cette mesure a été réalisée sur une moyenne d'un million de communications entre le nœud client et le serveur de sécurité. Nous obtenons une valeur moyenne de 260 μs pour le temps $temps_{comm_observateur2_rr}$, une valeur maximum de 400 μs et une valeur minimum 200 μs.

Lecture d'une trace avec une prise de décision Cette première mesure montre que la latence est générée, soit par la communication, soit par la prise de décision de la part de PIGA. Nous allons donc mesurer le temps de prise de décision de PIGA, $temps_{observateurs2_rr}$.

1. le proxy lit la trace en mémoire ;
2. la première mesure de temps est déclenchée ;
3. le proxy transmet la trace à PIGA.
4. PIGA prend une décision d'accès ;
5. le proxy récupère la décision ;
6. la seconde mesure de temps est réalisée.

Ainsi, en prenant le temps au niveau du proxy, nous pouvons déterminer le temps pris par PIGA pour prendre une décision d'accès. Cette procédure est illustrée par le diagramme de séquence 4.20.

On obtient un temps de prise de décision moyen pour PIGA de 220 μs, qui correspond au temps $temps_{observateurs2_rr}$. Tout comme pour la précédente mesure, ce temps est obtenu en faisant une moyenne sur un échantillon d'un million d'appels à PIGA.

Afin d'être précis, nous avons fait cette mesure au niveau du proxy qui est en C plutôt qu'au niveau de PIGA qui est en Java.

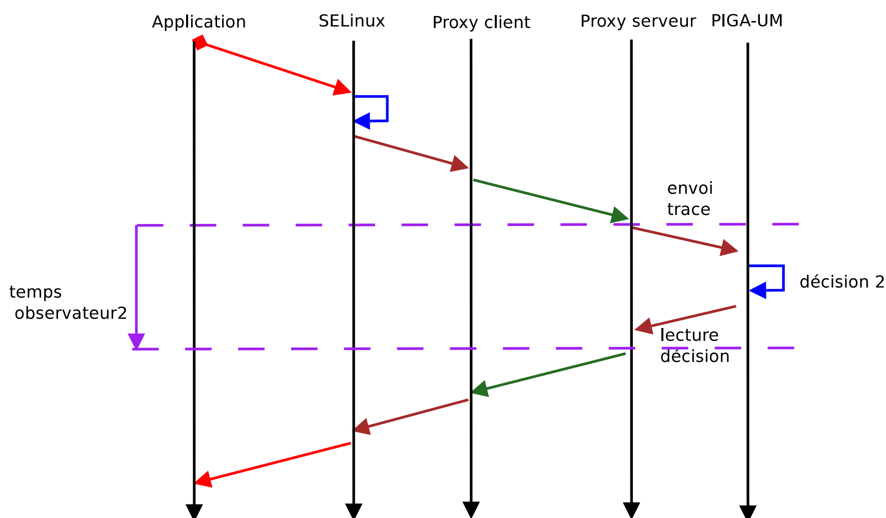

FIGURE 4.20 – Diagramme de séquence illustrant la mesure faite au niveau du proxy serveur pour calculer le temps de prise de décision de PIGA

Coût d'une communication À partir des deux dernières mesures, nous pouvons en déduire le coût d'une communication entre le proxy client et le proxy serveur.

En nous basant sur les moyennes obtenues, on obtient un temps $\frac{260-220}{2}$ soit 20 μs, correspondant au temps $temps_{communication}$. Les messages échangés entre les deux *proxys* ont une taille maximale de 300 octets.

Mesures spécifiques Nous avons dû réaliser des mesures spécifiques inhérentes à l'architecture que nous avons choisie et plus précisément, à la mise en place du proxy sur le nœud de calcul. En effet, de part sa conception, il doit récupérer les traces générées par PIGA-Kernel pour les envoyer sur le lien InfiniBand et il doit ensuite renvoyer à PIGA-Kernel les décisions d'accès prises par PIGA.

Pour ce faire, nous avons choisi d'utiliser le système de fichiers `procfs`. Nous avons donc deux *pipes* nommés : `pigaseq` et `piga_rep`. Le premier permet au proxy de lire les traces provenant de l'espace noyau tandis que le second lui permet d'écrire les décisions d'accès.

Nous avons donc dû mesurer le temps pris par le proxy pour lire et écrire dans ces *pipes*. Pour cela, nous avons daté l'envoi des traces ainsi que sa réception.

Le temps de lecture par le proxy donne, en moyenne, 10 μs. Par contre, la mesure de l'écriture se révèle beaucoup plus difficile à réaliser. Dans le but d'avoir une valeur correcte, nous allons prendre la valeur moyenne que nous avons calculée dans ce tableau 4.5, à savoir 115 μs.

4.3.2.4 Synthèse des mesures

Nous allons dans cette partie synthétiser les mesures que nous avons réalisées sur la répartition de deux observateurs au sein d'une architecture distribuée.

En nous appuyant sur le tableau 4.5, nous pouvons calculer que la latence générée en mode protection (ou requête/réponse) par l'architecture décentralisée de 417 μs.

Ce nombre est la somme de :

1. le temps de la prise de décision de la part de PIGA $temps_{observateurs2_rr}$: 220 μs ;
2. le temps de la communication (aller-retour) $2 * temps_{communication}$: 40 μs ;

3. le temps de lecture et d'écriture de la part du proxy dans les *pipes* pour communiquer avec PIGA-Kernel : $10 + 114\ \mu s$;
4. le temps introduit par SELinux $temps_{1_coloc_rr}$: $1\ \mu s$.

On obtient pour résultat 385 μs, soit une différence de 32 μs avec la mesure globale. Ce résultat s'explique par le fait que nous avons à chaque fois pris des moyennes sur un échantillon donné, donc il existe toujours une marge d'erreur non négligeable.

De plus, grâce à toutes ces différentes mesures, nous pouvons voir que les mécanismes de mesure de temps n'influent pas ou peu sur le temps global.

Nous proposons ce diagramme de séquence 4.21 qui nous permet de résumer de manière simple les différentes mesures que nous avons effectuées pour connaitre les points qui peuvent être améliorés en termes de performance. Ce diagramme illustre la répartition de SELinux avec PIGA en mode protection ($temp_{2_glob_dist_rr}$).

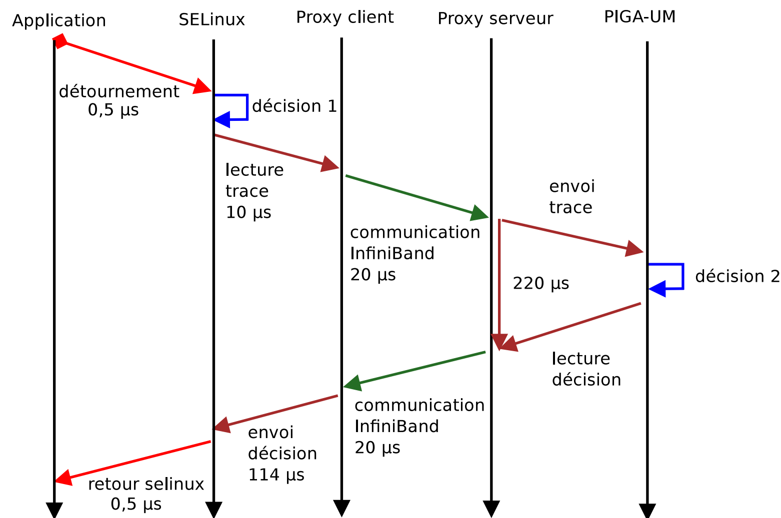

FIGURE 4.21 – Diagramme de séquence résumant les différentes mesures effectuées au sein de l'architecture distribuée

Nous proposons une synthèse des temps calculés dans le tableau suivant 4.6 établi par rapport aux résultats sur les expérimentations du stress du système de fichiers.

	temps par appel système en μs
$temps_{reference}$	114,3
SELinux $temps_{1_coloc_rr}$	115,3
SELinux + PIGA colocalisé : $temps_{2_glob_coloc_evi}$	631,12
SELinux + PIGA colocalisé : $temps_{2_glob_coloc_rr}$	1325,6
SELinux + PIGA distant : $temp_{2_glob_dist_evi}$	286,16
SELinux + PIGA distant : $temp_{2_glob_dist_rr}$	531
Communication + PIGA distant $temps_{comm_observateur2_rr}$	260
PIGA distant $temps_{observateur2_rr}$	220

TABLE 4.6 – Tableau résumant les mesures effectuées pour la répartition de deux observateurs

Nous proposons une synthèse des temps déduits dans le tableau suivant 4.7 établi par rapport aux résultats sur les expérimentations du stress du système de fichiers. Ce tableau montre que le déport du second moniteur de référence permet de réduire l'impact sur le nœud client. Ce gain se situe entre 2,2 et 2,5 par rapport au mode colocalisé.

	temps en μs
$temps_{communication}$	20
$temps_{latence_coloc_evi}$	517
$temps_{latence_coloc_rr}$	1211
$temps_{latence_dist_evi}$	172
$temps_{latence_dist_rr}$	417
$impact_{evi}$	2,2
$impact_{rr}$	2,5

TABLE 4.7 – Tableau résumant les temps calculés

Cette amélioration vient, d'une part par l'utilisation des technologies spécifiques à faibles latences des environnements HPC (InfiniBand et RDMA) et, d'autre part, par le déport de PIGA-UM. En déportant ce composant de notre architecture qui le plus consommateur en terme de ressources système (mémoire et processeur), nous réduisons son impact sur le nœud de calcul. De plus, comme PIGA-UM dispose d'une machine dédiée, il ne sera plus pré-empté par un autre processus et disposera de toute la puissance qui lui est nécessaire.

4.4 Amélioration de la sécurité

Dans cette partie, nous nous intéressons à l'attaque que nous avons présentée en introduction de ce mémoire1.3. Nous montrons comment nous contrôlons cette attaque grâce à l'utilisation des observateurs SELinux et PIGA. Pour pouvoir contrôler ce scénario complet, il est nécessaire tout d'abord de définir une propriété de sécurité qui sera appliquée par PIGA. Puis nous montrerons comme PIGA est capable de détecter ce scénario complet.

4.4.1 Propriété de sécurité

Nous montrons ici qu'il est possible d'utiliser PIGA pour améliorer la sécurité et traiter les scénarios complets d'attaque. À partir du scénario complet que nous avons étudié en introduction de ce mémoire (voir la section 1.3), nous allons exprimer une propriété de sécurité permettant de le contrôler.

Le listing 4.3 montre la définition de la propriété de sécurité que nous avons donnée à PIGA.

```
1 define exploit_staff_kernel( $sc1 IN CS, $sc2 IN CS, $sco1 IN CSO ) [
2      Foreach $eo1 IN is_execute_like(IS), Foreach $a1 IN ACT, Foreach $a2 IN
          ACT
3           TQ { ( $sc2 -> { $eo1 } $sco1 o $sc1 -> { $eo1 } $sco1 ) } ,
4               { not( exist() ) } ;
5 ];
```

Listing 4.3 – Propriété de sécurité contrôlant l'exploitation de la faille *perf_event*

Cette propriété se décrit de la manière suivante. Elle s'assure qu'il n'existe aucun lien *not(exist())* entre l'enchaînement des interactions suivantes : tout d'abord le contexte de sécurité $sc1 fait une opération d'exécution sur le contexte de sécurité $sco1 puis que le contexte de sécurité $sc2 réalise lui aussi une opération d'exécution sur le contexte de sécurité $sco1.

La fonction *not(exist())* s'assure qu'il n'y a aucun lien entre ces deux interactions. Ce lien peut être : un contexte de sécurité partagé, un processus en commun, une filiation, etc.

$$sc_1 \xrightarrow[execute]{} sco_1$$

$$sc_2 \xrightarrow[execute]{} sco_1$$

Nous avons appliqué cette propriété de la manière suivante 4.4.

```
1 exploit_staff_kernel( sc1:=staff_u:staff_r:staff_t,
2                  sc2:=system_u:system_r:kernel_t,
3                  sco1:=system_u:object_r:shell_exec_t );
```

Listing 4.4 – Application de la propriété de sécurité pour contrer l'exploitation de la faille

Le contexte de sécurité sc_1 correspond au contexte de sécurité de connexion de notre utilisateur de test. Ici nous avons choisi que ce dernier puisse être dans le rôle `staff_r`.

Le contexte de sécurité sc_2 correspond aux sujets qui s'exécutent en espace noyau. Or, comme la faille permet d'exploiter une faille au sein même du noyau, son contexte d'exécution sera `system_u:system_r:kernel_t`.

Le contexte de sécurité sco_1 correspond au contexte de sécurité des objets du système de type `shell`, comme `bash`, `zsh`, etc. Comme l'exploit lance un `shell` depuis l'espace noyau, nous avons décidé de bloquer explicitement cette exécution.

4.4.2 Application de la propriété de sécurité

Pour appliquer cette propriété, PIGA a besoin de la politique de contrôle d'accès de SELinux. L'objectif de PIGA est de trouver les chemins autorisés dans la politique permettant de violer la propriété de sécurité que nous venons de décrire.

À partir de cette propriété de sécurité seule et en parcourant la politique de SELinux, PIGA a détecté 6 activités pouvant mener à violer la propriété de sécurité.

Le listing 4.5 montre le résultat d'une détection de la part de PIGA lorsque l'utilisateur tente d'exploiter la faille.

```
1 ( system_u:system_r:kernel_t -( file { execute } )-> system_u:object_r:
    shell_exec_t rond staff_u:staff_r:staff_t -( file { execute } )-> system_u:
    object_r:shell_exec_t ) | 172.30.3.1 55031 system_u:system_r:kernel_t->
    system_u:object_r:shell_exec_t:file execute;2180 2081 -1 -1 exploit null
    138016
2 ( system_u:system_r:kernel_t -( file { execute_no_trans } )-> system_u:object_r
    :shell_exec_t rond staff_u:staff_r:staff_t -( file { execute_no_trans } )->
     system_u:object_r:shell_exec_t ) | 172.30.3.1 55034 system_u:system_r:
    kernel_t->system_u:object_r:shell_exec_t:file execute_no_trans;2180 2081 -1
     -1 exploit null 138016
```

Listing 4.5 – Détection par PIGA de la violation de la propriété de sécurité

Nous retrouvons dans cette détection les deux activités visées. La première concerne l'exécution par l'utilisateur du `shell` 4.6, puis nous avons une seconde activité qui détecte l'exécution du `shell` par le noyau 4.7.

```
1 staff_u:staff_r:staff_t -( file { execute } )-> system_u:object_r:shell_exec_t
    )
```

Listing 4.6 – Première activité détectée par PIGA

```
1 system_u:system_r:kernel_t -( file { execute } )-> system_u:object_r:
    shell_exec_t
```

Listing 4.7 – Deuxième activité détectée par PIGA

Nous sommes donc bien capables de détecter et de prévenir les scénarios d'attaque complets correspondant à de réelles vulnérabilités grâce à l'association de deux observateurs.

4.5 Discussion

Dans ce chapitre, nous avons proposé un modèle d'évaluation des performances pour la répartition des observateurs. Cette méthode permet de comparer les modes colocalisés et distants. Cette méthode conduit à comparer 8 mesures globales à un temps de référence sans observateur. Nous ajoutons à cette méthode des mesures détaillées qui permettent de connaître précisément les temps de communication entre les deux observateurs pour le mode distant ainsi que le temps de décision pour le second observateur.

Nous avons ensuite proposé une architecture originale pour déporter de manière efficace le second observateur. Nous avons pour cela exploité les technologies spécifiques des environnements HPC afin de minimiser les temps de communication. Nous avons mis en place cette architecture avec les moniteurs de référence SELinux et PIGA. Pour l'échange des requêtes, un proxy client et un proxy serveur permettent les communications sur des liens InfiniBand entre les deux moniteurs.

Nous avons enfin réalisé des expérimentations dans le but de comparer deux observateurs en mode colocalisé avec deux observateurs en mode distant. Nous avons pour cela appliqué notre méthode d'évaluation des performances sur l'architecture que nous avons mise en place. Durant ces

expérimentations, les deux moniteurs sont utilisés en mode requête/réponse et en mode évidence, ce qui conduit à devoir comparer 8 mesures avec un temps de référence.

Pour établir ces mesures, nous avons utilisé dans les premières expérimentations le logiciel `Linpack`, un logiciel spécialement utilisé dans les environnements HPC. Les résultats que nous avons obtenus montrent que la répartition des observateurs SELinux et PIGA n'impactent pas les performances du code de calcul car `Linpack` réalise essentiellement des calculs mémoire et peu d'entrées/sorties. On peut donc considérer que ce logiciel se place dans le meilleur des cas quant à l'impact des observateurs sur les performances. Dans la seconde expérimentation, nous avons voulu tester le pire des cas pour les observateurs en stressant le système de fichiers. Nous résumons les taux d'augmentation pour chaque temps global dans le tableau 4.8. Nous rappelons que dans les modes d'association colocalisés et distants, le premier observateur (SELinux) est toujours placé en mode requête/réponse.

Mesure	temps en μs	surcoût en %
$temps_{reference}$	114,3	
$temps_{1_coloc_evi}$	115,3	0.8
$temps_{1_coloc_rr}$	115,3	0.8
$temps_{2_glob_coloc_evi}$	631	453
$temps_{2_glob_coloc_rr}$	1325	1062
$temps_{2_glob_dist_evi}$	286	150
$temps_{2_glob_dist_rr}$	531	365

TABLE 4.8 – Tableau détaillant le surcoût de la répartition

Ce tableau montre que le mode distant améliore d'un facteur 3 les performances de la répartition des observateurs. Il est nécessaire de rappeler que ces chiffres sont donnés pour le pire des cas et qu'il n'y a pas de code de calcul qui réalise autant d'entrées/sorties lors de son exécution. A contrario, les résultats de `Linpack` représentent le meilleur des cas, puisque sur une durée de 23 heures, ce logiciel fait seulement 3000 opérations d'entrée/sortie. Nous pouvons dire qu'un code de calcul "standard" se situe entre les deux. De plus, nous avons volontairement choisi une configuration, au niveau des propriétés de sécurité PIGA, non optimale puisque nous garantissons l'intégrité et la confidentialité pour tout le système. Un gain important pourrait être obtenu en affinant les propriétés de sécurité PIGA et en évitant les propriétés contrôlant les flux indirects.

Dans la pratique, un code de calcul fait des entrées/sorties sur le système de fichiers essentiellement pour réaliser des `checkpoints` (points de sauvegarde en cas de panne). Ces entrées/sorties se traduisent pas l'écriture (ou la lecture) de fichiers ayant une taille importante mais générant peu d'appels, donc les moniteurs sont moins sollicités que dans notre scénario de stress. Par conséquent, le comportement d'un code de calcul sera plus proche du comportement de `Linpack` que de notre logiciel pour stresser les moniteurs de référence.

Nous notons que le logiciel `Linpack` réalise environ 3000 appels système en 23 heures, soit 0,036 appels système par secondes (soit 2,17 par minutes) alors que notre logiciel de stress réalise 5 000 000 d'appels système en 1 heure, soit 1 388 appels système par secondes. Dans le cas de `Linpack`, nous ne notons aucune augmentation de la latence tandis que l'impact du stress du système de fichiers est compris entre 150 et 365 % pour le mode distant, que ce soit en mode évidence ou requête/réponse. Cependant, aucun de ces deux logiciels ne correspond exactement au nombre d'entrées/sorties d'un code de calcul.

Par une estimation linéaire, nous pouvons estimer qu'un code de calcul réalise entre 100 et 200 appels système par secondes, nous pouvons estimer l'impact suivant les modes de répartition :

— en mode évidence (détection) et distant, l'impact sur les performances du code de calcul serait compris entre 10 et 20%. Puisqu'un ordre de 1500 appels système par secondes

correspond (environ) à un impact de 150%, on obtient par estimation linéaire un impact de 150/1500x100 soit 10% pour 100 appels système par secondes.

— en mode requête/réponse (protection) et distant, l'impact sur les performances du code de calcul sera entre 25 et 50% par estimation linéaire.

Pour l'instant, il semble difficile de mettre en place une répartition de deux observateurs tels que SELinux et PIGA sur les nœuds de calcul sans nuire aux performances. Le gain réalisé par le déport du second moniteur en mode détection introduisant un surcoût de l'ordre de 10%. Néanmoins, l'amélioration des politiques PIGA devrait permettre de passer à un surcoût plus faible, potentiellement de l'ordre de 5% selon les propriétés requises. D'autres améliorations sur le moteur de PIGA devrait permettre d'aller encore en dessous.

Les autres nœuds du calculateur sont moins demandeurs en termes de performance. Nous pouvons par exemple citer les nœuds de visualisation ou de connexion.

Les nœuds de visualisation sont utilisés pour présenter graphiquement les résultats obtenus lorsque les calculs sont finis. Sur ce type de nœuds, les programmes chargent des fichiers de taille importante (plusieurs téraoctets) mais réalisent peu d'appels système. Par conséquent, l'impact devrait être acceptable pour la visualisation.

Pour les nœuds de connexion, ils sont utilisés pour accéder au calculateur, ce sont donc des nœuds exposés aux attaques. En effet, les utilisateurs disposent d'un accès `ssh` leur permettant de déposer leur code de calcul, tester leur code, etc. Sur ces types de nœuds, les besoins en performances sont moins importants que sur les nœuds de calcul mais les besoins en sécurité sont donc plus importants. Ce type de nœuds est donc tout à fait candidat pour utiliser nos deux moniteurs répartis en mode protection.

Enfin, nous avons illustré l'efficacité de deux moniteurs pour prévenir les scénarios complets d'attaque. Pour la vulnérabilité réelle décrite en introduction de ce mémoire, nous avons défini une propriété de sécurité qui prévient le scénario de façon optimiste c'est-à-dire qui laisse les activités légitimes avancer. Il est à noter que la propriété est générique car elle prévient toutes les vulnérabilités de ce type.

Chapitre 5

Contrôle d'accès obligatoire pour Windows et Expérimentations

Dans l'état de l'art 2.3.5, nous avons montré la nécessité d'avoir un moniteur de référence pour les systèmes Windows. Dans le chapitre 3, nous avons défini un modèle de moniteur pour contrôler les interactions directes. Nous proposons dans ce chapitre une implantation fonctionnelle d'un tel moniteur pour les systèmes Windows.

Un moniteur de référence doit mettre en place des mécanismes de détournement du flux d'exécution au sein du système, afin d'appliquer les décisions prises par la partie moniteur de référence de notre implantation.

Notre moniteur de référence implante les deux modèles de protection obligatoires que nous avons présentés dans la section 3.2, à savoir les modèles PBAC et DTE. Pour obtenir une politique PBAC portable sur les différents systèmes Windows, nous apportons une réponse à la problématique de désignation des ressources. Pour le modèle DTE, nous proposons de calculer dynamiquement les contextes de sécurité à chaque interaction.

Toutefois, l'écriture d'une politique de sécurité pour tout un système est une tâche complexe et laborieuse et nous avons donc mis en place un mécanisme facilitant leur écriture. Ce mécanisme se base sur les traces générées par l'observateur pour dériver, de manière automatique, les règles correspondantes.

Ce chapitre est divisé en quatre parties. Dans une première partie, nous traiterons de la problématique de création des politiques. Nous proposerons une solution pour résoudre la problématique de la construction des noms symboliques absolus indépendants de la localisation. Nous présenterons enfin notre mécanisme de génération automatique de la politique.

La seconde partie sera consacrée à l'implantation de deux *Policy Enforcement Point*, c'est-à-dire des méthodes pour détourner le flux d'exécution régulier du système, que nous avons décrites dans la partie 3.3. Nous avons ici choisi de nous focaliser sur le détournement de la table des appels système et sur la création d'un *filter-driver*. Nous expliquerons en détail le fonctionnement de ces deux méthodes.

La troisième partie traitera des expérimentations que nous avons réalisées. Dans une première section, nous montrerons que nous sommes capables de bloquer des attaques simples visant à violer une propriété de sécurité telle que la confidentialité. Nous illustrerons les limites de ce mécanisme grâce à des scénarios d'attaques complets. Pour franchir ces limites, nous introduirons l'utilisation d'un second moniteur, PIGA.

Dans une seconde section, nous proposerons des modifications de notre moniteur pour l'orienter sur l'étude des logiciels malveillants. Nous présentons les modifications apportées ainsi que la plateforme mise en place pour faciliter l'interprétation des résultats. Pour cela, nous reprendrons l'exemple que nous avons présenté dans l'introduction de ce mémoire 1.3.

Enfin, la dernière partie discutera les choix que nous avons faits ainsi que les limites de notre implantation.

L'implantation de la solution basée sur le modèle DTE et sur le détournement de la table des appels système a fait l'objet d'une publication [Gros *et al.*, 2012]. L'application des propriétés de sécurité est détaillée dans [Blanc *et al.*, 2012]. Une application du modèle de protection a été proposée au sein des *clouds* dans [Blanc *et al.*, 2014].

5.1 Politique de contrôle d'accès direct

Cette section s'intéresse à la création et à la gestion des politiques de contrôle d'accès direct. Nous traitons dans cette partie les deux modèles de protection, PBAC dans un premier temps puis le modèle DTE dans un second temps.

Chaque sous-section est organisée ainsi : dans une première partie, nous expliquerons les choix réalisés pour représenter les ressources du système. Puis, nous aborderons la création d'une politique de contrôle d'accès s'appliquant à chaque modèle de protection. Enfin, nous détaillerons le mécanisme d'automatisation pour générer des politiques de sécurité.

5.1.1 *Path-Based Access Control*

Représentation du système

Nous avons détaillé dans la partie 3.2.1.1 la problématique du système de nommage des fichiers sous Windows. Pour résoudre ce problème, nous proposons l'utilisation de noms symboliques absolus indépendants de la localisation.

L'implantation choisie se base sur l'utilisation des variables d'environnement présentes sous les systèmes Windows. Les variables d'environnement dites "système", c'est-à-dire gérées directement par le noyau, sont communes à tous les systèmes Windows. On peut par exemple citer `systemroot` qui est la variable d'environnement pointant sur le répertoire d'installation du dossier `Windows`. Pour simplifier leur utilisation, nous utilisons les dénominations utilisées en *batch*, ce qui se traduit par l'utilisation de la syntaxe `%systemroot%`.

En utilisant les variables d'environnement, nous sommes capables de générer des politiques de sécurité portables sur chaque système Windows. Par exemple, le fichier `C:\Windows\system32\cmd.exe` se transforme en `%systemroot%\System32\cmd.exe`.

Cette méthode fonctionne aussi bien pour les sujets que pour les objets de type fichier. Les répertoires sont traités de la même façon. Tous les répertoires système possèdent une variable d'environnement associée. Lorsqu'un répertoire est au niveau du point de montage du système mais ne possède pas de variable d'environnement associée, il suffit d'utiliser la variable définie pour le point de montage du système `%systemdrive%`. Pour étendre ce modèle aux disques ou aux partitions qui seraient montés sur un autre lecteur et qui ne possèderaient pas de variables associées, il est nécessaire d'en créer de nouvelles. Concrètement, si une partition est montée sur la lettre `D:` et ne contient aucun dossier système comme les répertoires des utilisateurs ou des programmes, il est nécessaire de la nommer avec une variable comme `%datadrive%`.

La gestion du registre est simplifiée par rapport à celle du système de fichiers. Les ruches, qui sont les éléments qui hébergent les clés du registre, sont identiques sur tous les systèmes Windows. On peut donc s'en servir comme espace de noms pour désigner les clés. Ainsi, la clé *CurrentVersion* qui est dans la ruche `HKEY_LOCAL_MACHINE`, est représentée par le chemin `\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion`.

Les éléments du réseau se gèrent aussi de façon naturelle. Ils peuvent être identifiés à la fois par un couple d'adresse IP et de port, soit par le nom des hôtes. Pour qu'il n'y ait pas de confusion

entre un objet local et un objet réseau, nous les préfixons par l'espace de nom `RES`. Ainsi l'objet 192.168.1.254 avec le port 80 est représenté par la directive `RES/192.168.1.254 80`. Nous avons la même directive pour les noms d'hôte, par exemple `RES/www.google.fr 80`.

Extrait d'une politique basée sur les chemins complets

Nous avons précisé dans le listing 3.9 les terminaux propres au modèle de protection PBAC. De plus, la définition de la grammaire générique 3.1 précise la forme des vecteurs d'accès que nous reprenons dans ce listing 5.1 :

```
1 vecteur_acces : contexte Delimiteurs_debut? (liste_operations_elementaires)+
      Delimiteurs_fin? ;
```

Listing 5.1 – Règle générique d'un vecteur d'accès extrait de la grammaire générique

Donc à partir de ces différents éléments, nous allons pouvoir écrire une politique de sécurité basée sur le modèle de protection PBAC.

Ainsi, le listing 5.2 illustre la manière d'autoriser le sujet `%systemroot%\explorer.exe` à lire (`r`) et exécuter (`x`) l'objet ayant pour chemin `%systemroot%\System32\cmd.exe`. Ce fichier se situe donc dans le répertoire d'installation du dossier `Windows`. Le sujet `%systemroot%\explorer.exe` est aussi autorisé à aller lire (`r`) et écrire (`w`) dans le fichier `%programfiles%\fichier.txt`, dont la variable d'environnement correspond au répertoire nommé `Program Files`.

```
1 %systemroot%\explorer.exe {
2       %systemroot%\System32\cmd.exe rx
3       %programfiles%\fichier.txt rw
4 }
```

Listing 5.2 – Extrait d'une politique de type TPE

Pour les accès au registre, les règles s'écrivent de la même façon. Ainsi le listing 5.3 montre comment autoriser le sujet `%systemroot%\explorer.exe` à lire (`r`) la clé `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion`.

```
1 %systemroot%\explorer.exe {
2       HKEY_LOCAL__MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion r
3 }
```

Listing 5.3 – Extrait d'une politique de type PBAC pour le registre

Pour les autorisations liées aux objets du réseau, nous écrivons la politique suivante 5.4. Ainsi la première ligne autorise le sujet `%systemroot%\explorer.exe` à se connecter (`c`) au domaine `www.google.fr` vers le port 80.

```
1 %systemroot%\explorer.exe {
2       RES\www.google.fr 80 c
3 }
```

Listing 5.4 – Extrait d'une politique de type PBAC pour le réseau

Création d'une politique basée sur le modèle PBAC

La création d'une politique au format PBAC peut se faire de manière manuelle mais aussi automatique. Ces deux méthodes peuvent se faire en se basant sur une trace d'audit. Cette méthode simplifie l'écriture des règles d'accès.

Pour faciliter l'administration d'une politique de contrôle d'accès basée sur le modèle PBAC, nous proposons l'utilisation d'un outil capable d'automatiser la création d'une telle politique en utilisant les fichiers d'audit générés par le moniteur de référence. Nous présentons la méthode d'automatisation plus en détail dans la section 5.1.4.

Ainsi, à partir du listing 5.5, nous avons généré la règle 5.6.

```
type=PBAC audit(1174358746:36452) ips:denied { read execute } for pid=2004 comm
    ="%systemroot%\explorer.exe" ppid=872 path="%systemroot%\system32\audiodev.
    dll" tclass=file
```

Listing 5.5 – Trace d'audit pour l'implantation utilisant le modèle PBAC

```
subjet %systemroot%\explorer {
     %systemroot%\system32\audiodev.dll rwa
}
```

Listing 5.6 – Règle de contrôle d'accès pour le modèle PBAC résultat de l'apprentissage

Une fois que la règle a été ajoutée à la politique courante et que l'observateur a rechargé cette nouvelle politique, lorsque le moniteur observe que l'interaction courante est présente dans la politique, il va le spécifier dans la trace qu'il génère. Il faut noter que, mis à part le `timestamp` et l'identifiant de trace qui sont modifiés, la seule différence se trouve dans la décision prise par l'observateur qui passe de `denied` à `granted` comme le montre les traces 5.5 et 5.7

```
type=PBAC audit(1174956145:264) ips:granted { read execute } for pid=1024 comm=
    "%systemroot%\explorer.exe" ppid=988 path="%systemroot%\system32\audiodev.
    dll" tclass=file
```

Listing 5.7 – Trace d'audit pour l'implantation utilisant le modèle PBAC

Trusted Path Execution

Nous avons défini les répertoires de confiance à partir d'une installation classique d'un système Windows. Ainsi, les répertoires `%systemroot%` et `%systemroot%\System32` qui sont les répertoires d'installation du système, et `%programfiles%`, qui est le répertoire d'installation des applications tierces, sont des répertoires de confiance. Naturellement, pour qu'un fichier puisse être exécuté, il est nécessaire qu'une règle d'accès autorise un sujet à l'exécuter. La définition des répertoires de confiance permet de mettre en place une règle d'accès simple qui stipule que tout sujet qui souhaite exécuter un fichier qui n'est pas dans un de ces dossiers, verra son accès refusé.

Les répertoires d'où sont exécutés les exécutables que l'administrateur autorise, sont notés en début de politique sous la forme suivante 5.8 :

```
tpe {
   %programfiles%
   %systemroot%
}
```

Listing 5.8 – Définition des répertoires de confiance

5.1.2 Domain and Type Enforcement

Labellisation

Le second modèle de protection que nous avons traité est le modèle DTE. Nous avons présenté les avantages d'utiliser ce modèle de protection dans la section 3.2.2.2.

Sur les systèmes Linux supportant SELinux, les contextes de sécurité des objets sont stockés dans les attributs étendus du système de fichiers. Ces informations sont disponibles en utilisant les commandes comme `ls -Z` ou encore `getfattr`. Un exemple de résultat est montré dans le listing 5.9. Il est possible de stocker ces informations car le système de fichiers prévoit la possibilité d'avoir des noms arbitraires d'attributs étendus. L'avantage de cette technique est la persistance des contextes mais celle-ci pose des problèmes pratiques.

```
getfattr -m security.selinux -d pol.xml.bak
# file: pol.xml.bak
security.selinux="system_u:object_r:default_t:s0"
```

Listing 5.9 – Récupération des attributs étendus sous Linux

Cependant, dans le cas de Windows, il n'est pas prévu d'étendre le système de fichiers pour ajouter des contextes de sécurité comme sous Linux car c'est un système de fichiers propriétaire (NTFS). Par conséquent, plutôt que de les stocker, nous avons fait le choix de les calculer dynamiquement. Cette opération se déroule durant la phase de prétraitement du moniteur de référence (phase 2.1 de la figure 3.9).

Le calcul à la volée des contextes de sécurité implique une considération importante : il est nécessaire de pouvoir identifier précisément chaque ressource. Sous SELinux, le fait d'avoir des contextes de sécurité non uniques est compensé par l'impossibilité pour un utilisateur de modifier les contextes. Ainsi, SELinux s'appuie à la fois sur le nom de la ressource mais aussi sur son contexte pour différencier les objets sur le système.

Le calcul à la volée des contextes ne permet pas de faire cette distinction dans ce cas. Par exemple, il n'est pas possible de différencier un fichier `explorer.exe` qui se trouve dans `C:\Windows\explorer.exe` et dans `C:\Users\Damien\explorer.exe`, lorsque les contextes de sécurité ne sont basés que sur le nom relatif des fichiers.

Pour résoudre ce problème, nous avons utilisé notre approche de désignation des ressources déjà appliquée au modèle PBAC. Ainsi, nous offrons une méthode dynamique et portable. Nous ajoutons une couche d'abstraction par l'utilisation des noms symboliques absolus indépendants de la localisation. Dans ce cas, nous avons aussi utilisé les variables d'environnement puisqu'elles sont communes à tous les systèmes Windows.

À partir de ces choix, nous pouvons labelliser le système. Nous rappelons qu'un contexte de sécurité est constitué d'une identité, d'un rôle, d'un type pour les objets et d'un domaine pour les sujets. Dans l'implantation actuelle, nous n'utilisons ni l'identité, ni les rôles. Nous nous sommes concentrés sur les types et les domaines tout comme le fait SELinux.

Les objets Dans la section 3.2.2.2 où nous avons présenté notre formalisation pour la description des objets dans le modèle de protection DTE, nous avons défini des catégories pour identifier la nature des ressources. Par exemple, un répertoire a un type associé finissant par `dir_t`.

Le listing 5.10 exprime une règle de correspondance entre le chemin exprimé sous la forme de variable d'environnement et le contexte de sécurité qui lui est associé. Nous noterons que tous les objets possèdent les mêmes identité et rôle `system_u:object_r`. Le rôle `object_r` précise explicitement que ce contexte de sécurité se rattache à un objet du système.

```
%programfiles% system_u:object_r:programfiles_dir_t
```

Listing 5.10 – Contexte de sécurité associé au répertoire *Program Files*

Le contexte de sécurité des objets calculé à la volée fait apparaître son chemin complet dans le système de fichiers. Ainsi, si l'objet courant est dans un point de l'arborescence avec plusieurs répertoires parents, son contexte contiendra tous ces répertoires. Pour aider à la lecture des contextes de sécurité dans les fichiers d'audit, nous séparons chaque répertoire par

le caractère |. Ce caractère étant un caractère interdit dans le nom des fichiers et des répertoires sous Windows, nous sommes sûrs qu'il n'apparaîtra pas dans le nom des objets. Par exemple, le répertoire `C:\Program Files\Mozilla\Plugins` a pour contexte de sécurité `system_u:object_r:programfiles|mozilla|plugins_dir_t`.

La construction du contexte de sécurité pour les fichiers se fait de façon similaire. Cependant, un répertoire possède une catégorie spécifique aux répertoires, c'est-à-dire un suffixe `dir_r` ajouté au type de son contexte de sécurité, il faut aussi définir les catégories pour les fichiers. Pour faciliter la reconnaissance faite par le moniteur de référence, il va se baser sur les extensions des fichiers pour en extraire leur catégorie.

Nous obtenons un grand nombre de catégories possibles pour les fichiers : exécutable, bibliothèque, *driver*, un document *Word*, etc. Dans ces catégories, nous portons une attention toute particulière à un sous-ensemble : les fichiers qui peuvent être exécutés comme les binaires exécutables mais aussi les bibliothèques, ainsi que les fichiers pouvant transporter une charge virale tels que les fichiers `pdf` ou `MS Word`.

Les fichiers ont aussi un suffixe spécifique dans leur type qui représente leur catégorie. Ainsi, un fichier exécutable, classiquement représenté par son extension `.exe`, aura comme type `exec_t`. Un *driver*, qui a pour extension `.sys` aura comme type `sys_t`.

Ainsi, le fichier `C:\Windows\system32\cmd.exe` a pour contexte de sécurité `system_u:object_r:systemroot|system32|cmd_exec_t`. De manière analogue, les fichiers de bibliothèques dynamiques (`dll`) ont un type se finissant par `lib_t`.

Grâce à cette méthode de labellisation dynamique, chaque objet du système possède un contexte de sécurité précis basé sur sa place dans l'arborescence du système de fichiers. Cependant, cette méthode conduit à avoir potentiellement un nombre important de contextes de sécurité. Nous avons donc prévu un traitement spécifique pour certains répertoires dans lesquels fichiers et répertoires ont tous le même contexte de sécurité.

Cette exception dans notre méthode de labellisation dynamique s'applique essentiellement pour les dossiers temporaires. En effet, il n'est pas possible de prévoir le contenu de ce type de dossiers dans le but de pouvoir faire une politique de contrôle d'accès. Prenons un répertoire temporaire tel que `C:\Temp`, si un utilisateur y crée un fichier, alors il aura pour contexte de sécurité `system_u:object_r:tmp_t`.

Les sujets Les contextes de sécurité sujet sont construits à partir du nom relatif du fichier du binaire exécuté. Les identités et rôles utilisés sont génériques pour l'instant. Par exemple, le processus `explorer.exe` a pour contexte de sécurité le label `system_u:system_r:explorer_t`.

Il n'est pas nécessaire de stocker tout le chemin du binaire dans le contexte de sécurité sujet. L'objectif est d'alléger les traces et les politiques, ainsi que de faciliter leur lecture. Cependant, nous pourrions tout à fait stocker tout le chemin dans le contexte de sécurité comme pour les objets.

Le registre Dans l'implantation proposée, nous ne faisons qu'une labellisation générique du registre. Cela signifie que tous les objets du registre ont pour contexte de sécurité `system_u:object_r:registry_t`.

Néanmoins, à partir de la formalisation que nous avons proposée, nous pouvons définir les contextes de sécurité pour les objets du registre. Ainsi, en utilisant les mêmes méthodes que pour le modèle PBAC, c'est-à-dire en utilisant le nom de la ruche comme `namespace`, on peut construire les contextes de sécurité en utilisant tout le nom de la clé. On utilisera aussi le même séparateur que celui utilisé pour les chemins dans le nom des objets, à savoir |, pour décrire l'arborescence de la clé.

On utilisera aussi des catégories spécifiques. Ainsi, une clé aura pour suffixe key_t, alors qu'une valeur aura pour catégorie value_t. Le listing 5.11 montre sur deux exemples le résultat sur une clé et sur une valeur.

```
1 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run system_u:
    object_r:hkey_local_machine|software\microsoft|windows|currentversion|
    run_key_t
2 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run|Avira system_u
    :object_r:hkey_local_machine|software\microsoft|windows|currentversion|
    run_avira_value_t
```

Listing 5.11 – Exemples d'associations d'éléments du registre et de contextes de sécurité

Extrait d'une politique basée sur les types

Comme pour le modèle de protection PBAC, nous avons formalisé les terminaux pour le modèle de protection DTE dans le listing 3.12. La spécification des terminaux nous permet de définir de manière explicite les traitements que l'observateur doit réaliser lorsqu'il intercepte les interactions directes.

Les règles d'autorisation sont plus complexes que dans le cas de PBAC car les permissions exprimées sont basées sur les appels système, donc avec une plus grande finesse. Ainsi, pour autoriser (allow) un sujet ayant comme domaine explorer_t à exécuter un invite de commande, il faut, tout d'abord, l'autoriser à récupérer des informations et à lire le fichier binaire. Cette règle est représentée par la ligne 1 du listing 5.12. Ensuite on doit l'autoriser à exécuter le fichier binaire pour en faire un processus, ce qui est représenté pour la ligne 2 du listing.

```
1 allow explorer_t systemroot|system32|cmd_exec_t:file { read execute create
    getattr }
2 allow explorer_t systemroot|system32|cmd_exec_t:process { execute }
```

Listing 5.12 – Extrait d'une politique SEWindows

Lors de l'exécution d'un fichier binaire (généralement un .exe), il se produit une transition du type du fichier vers un domaine. Par exemple, lorsque le sujet ayant comme domaine explorer_t exécute un invite de commande, soit le binaire C:\Windows\system32\cmd.exe qui a pour contexte de sécurité objet system_u:object_r:%systemroot%|system32|cmd_exec_t, le nouveau processus transite vers le contexte de sécurité sujet system_u:system_r:cmd_t.

Le contrôle des accès pour le registre se fait de manière similaire. Il est possible d'associer chaque opération du registre à une opération élémentaire (lire, écrire, exécuter, etc). Donc la création d'une règle d'accès pour le registre se fait comme une règle pour le système de fichiers. Le listing 5.13 détaille une règle pour la lecture du contenu d'une clé.

```
1 allow explorer_t hkey_local_machine|software|microsoft|windows|currentversion|
    run_key_t:key { read execute }
```

Listing 5.13 – Extrait d'une politique SEWindows pour la gestion du registre

Nous avons défini dans la grammaire trois traitements possibles pour une règle d'accès. L'instruction allow autorise de manière explicite les interactions. L'instruction allowaudit va générer un message dans le fichier d'audit spécifiant que cette règle a été utilisée. Elle peut être utilisée pour surveiller des interactions particulières. Toutefois elle n'autorise pas l'interaction, pour cela, elle doit être complétée avec une règle allow. Enfin, l'instruction neverallow empêche explicitement l'interaction décrite par la règle d'accès. Cette règle prime sur toute règle allow autorisant la même interaction.

Création de la politique basée sur le modèle DTE

Comme pour le modèle de protection PBAC, la création d'une politique pour le modèle DTE peut se faire de manière manuelle ou automatique. Nous allons ici décrire la création de règles d'accès à partir d'une trace générée par le moniteur.

Nous allons détailler cette création sur un exemple concret. Nous avons lancé l'application `Windows Media Player`. Au niveau du mécanisme de création de processus de Windows, c'est le processus `%systemroot%\explorer.exe` qui lance le processus qui a pour nom `wmplayer.exe`. Cette opération se traduit par la génération de plusieurs traces. Nous avons extrait deux traces pour illustrer notre propos.

Dans un premier temps, le processus `%systemroot%\explorer.exe` va demander les opérations élémentaires `{ read execute create getattr }` sur le fichier `%programfiles%\windows media player\wmplayer.exe` comme le montre la première ligne du listing 5.14.

Le processus de labellisation transforme les noms et chemins des fichiers en contexte de sécurité comme expliqué dans la partie 5.1.2 . Ainsi, le processus `%systemroot%\explorer.exe` aura pour contexte de sécurité sujet `system_u:system_r:explorer_t`. De la même façon, l'objet `%programfiles%\windows media player\wmplayer.exe` aura pour contexte de sécurité `system_u:system_r:programfiles|windowsmediaplayer|wmplayer_exec_t`.

Avec ces éléments, nous sommes capables de créer une première règle pour notre politique de contrôle d'accès comme le montre la première règle du fichier de politique présent en listing 5.15.

Ensuite, il y a une deuxième interaction faite par le sujet qui est la création du processus. Cette interaction se traduit au niveau du moniteur par une seconde trace. La différence se fait sur la classe de l'objet. En effet, sur la première trace, l'interaction se fait sur le fichier, alors que la deuxième trace concerne une création de processus, c'est-à-dire à la transition du fichier vers son domaine.

Pour autoriser la création du processus, il est nécessaire d'ajouter la deuxième ligne présente dans le listing 5.15.

```
1 type=DTE audit(1285242977:13) avc:denied { read execute create getattr } for
    pid=1576 comm="%systemroot%\explorer.exe" ppid=1528 path="%programfiles%\
    windows media player\wmplayer.exe" scontext=system_u:system_r:explorer_t
    tcontext=system_u:object_r:programfiles|windowsmediaplayer|wmplayer_exec_t
    tclass=file
2 type=DTE audit(1285242977:14) avc:denied { execute } for pid=1576 comm="%
    systemroot%\explorer.exe" ppid=1576 path="%programfiles%\windows media
    player\wmplayer.exe" scontext=system_u:system_r:explorer_t tcontext=
    system_u:object_r:programfiles|windowsmediaplayer|wmplayer_exec_t tclass=
    process
```

Listing 5.14 – Extrait d'un fichier d'audit

```
1 allow explorer_t programfiles|windowsmediaplayer|wmplayer_exec_t:file { read
    execute create getattr }
2 allow explorer_t programfiles|windowsmediaplayer|wmplayer_exec_t:process {
    execute }
```

Listing 5.15 – Règles de contrôle d'accès créées à partir d'une trace

5.1.3 Modes de fonctionnement

Notre moniteur de référence a trois modes de fonctionnement : *disabled*, *permissif* (évidence) aussi appelé détection et *enforcing* ou protection (requête/réponse). Dans le mode *disabled*, le flux

d'exécution n'est pas détourné par le moniteur et aucune décision n'est calculée. Nous allons donc détailler dans cette section les deux autres modes du moniteur.

Mode détection (Évidence)

Le mode détection, ou *permissif*, est un mode dans lequel le moniteur n'applique pas sa décision. Le flux d'exécution est détourné, une décision est bien calculée au regard de la politique, mais elle n'est pas appliquée. Ce mode peut aussi fonctionner sans aucune politique prédéfinie, dans ce cas, toutes les interactions seront considérées comme refusées. Cette particularité peut se révéler intéressante pour l'étude des logiciels malveillants puisqu'il sera alors possible d'enregistrer toutes les opérations qui se déroulent au sein du système.

Ce mode a deux objectifs : la première, il permet de créer une politique de contrôle d'accès pour un nouveau programme et la seconde, il permet de valider la politique. Ce mode peut aussi être appelé `mode d'apprentissage` quand il est utilisé pour créer une politique.

La création de la politique se fait à partir des traces générées par le moniteur. Une fois les traces générées, l'écriture de la politique peut se faire de manière manuelle ou automatique comme nous le détaillons dans la section 5.1.4. Chaque interaction est vérifiée dans la politique et une trace est générée spécifiant si cette interaction est autorisée, c'est-à-dire qu'il existe une règle d'accès dans la politique qui l'autorise, sinon l'interaction est refusée.

Le mode apprentissage peut être réalisé avec ou sans politique de sécurité. Si une politique existe, les traces contiennent le terme `granted` stipulant que l'interaction est autorisée dans la politique courante comme le montre le listing 5.16. La règle pour autoriser cette interaction a été définie dans le listing 5.12.

```
1 type=DTE audit(1285243100:4600) avc:granted { execute } for pid=1576 comm="%
    systemroot%\explorer.exe" ppid=1576 path="%systemroot%\system32\cmd.exe"
    scontext=system_u:system_r:explorer_t tcontext=system_u:object_r:systemroot
    |system32|cmd_exec_t tclass=process
```

Listing 5.16 – Extrait d'un fichier d'audit autorisant une interaction

Une fois que l'administrateur considère avoir réalisé toutes les interactions nécessaires pour que son programme puisse fonctionner et avant de passer le moniteur en mode protection, ce qui pourrait bloquer la machine, il peut valider sa politique dans le mode détection en vérifiant les traces générées.

Ce mode peut aussi se révéler utile dans des environnements où seule la détection des interactions est voulue.

Mode protection (Requête/Réponse)

Le mode protection, ou *enforcing*, est le mode dans lequel le moniteur applique les décisions. Si une interaction n'est pas dans la politique de contrôle d'accès, alors elle sera bloquée par le moniteur et une trace sera générée pour notifier l'administrateur de ce refus.

5.1.4 Définition des politiques

Les politiques de contrôle d'accès peuvent être écrites soit de façon manuelle, soit de manière automatique grâce à l'outil que nous proposons. Nous allons dans cette section détailler les deux fonctionnements.

Automatisation

Nous proposons un outil pour la création automatique d'une politique de contrôle d'accès, quel que soit le modèle de protection. Grâce à cette automatisation, la politique s'écrit par répétition.

Pour réaliser cet apprentissage, il est nécessaire de mettre le moniteur en mode détection, ainsi le programme ou processus pour lequel on veut créer la nouvelle politique ne sera pas bloqué.

Prenons un exemple illustré par la figure 5.1. Nous désirons ici ajouter un programme sur notre système d'exploitation, il faut donc ajouter les règles d'accès correspondantes pour qu'il puisse fonctionner correctement. Ce mode peut fonctionner avec ou sans politique de base. Si le moniteur n'a pas de politique de base, il va enregistrer toutes les interactions faites sur le système, même celles qui ne sont pas réalisées par le programme que l'on veut ajouter.

La méthodologie d'apprentissage est la suivante. On change le mode du moniteur en détection. On lance une première fois le nouveau programme (flèche 0 du schéma). Le moniteur trace les interactions du processus (flèche 1) et les enregistre dans le fichier d'audit (flèche 2). Notre outil parcourt le fichier d'audit (flèche 3) et transforme automatiquement les traces en règles d'accès au bon format, c'est-à-dire PBAC ou DTE. Ces nouvelles règles sont ajoutées à la politique courante (flèche 4) puis cette nouvelle politique est rechargée par le moniteur (flèche 5). Le processus ou programme que l'on veut ajouter est ensuite relancé.

La fin du processus d'apprentissage est laissée à la discrétion de l'administrateur. En effet, il n'est pas possible de déterminer de manière certaine quand un programme a réalisé toutes les interactions de manière automatique. Par exemple, pour une application graphique, il faut que l'administrateur réalise lui-même les tâches qu'il souhaite autoriser. Il est donc conseillé d'avoir une `check-list` précisant les tâches nécessaires au moment de lancer l'apprentissage.

L'apprentissage peut être considéré comme terminé lorsqu'on a réalisé toutes les tâches listées dans la `check-list` et qu'il n'apparaît plus aucune trace d'accès bloqué.

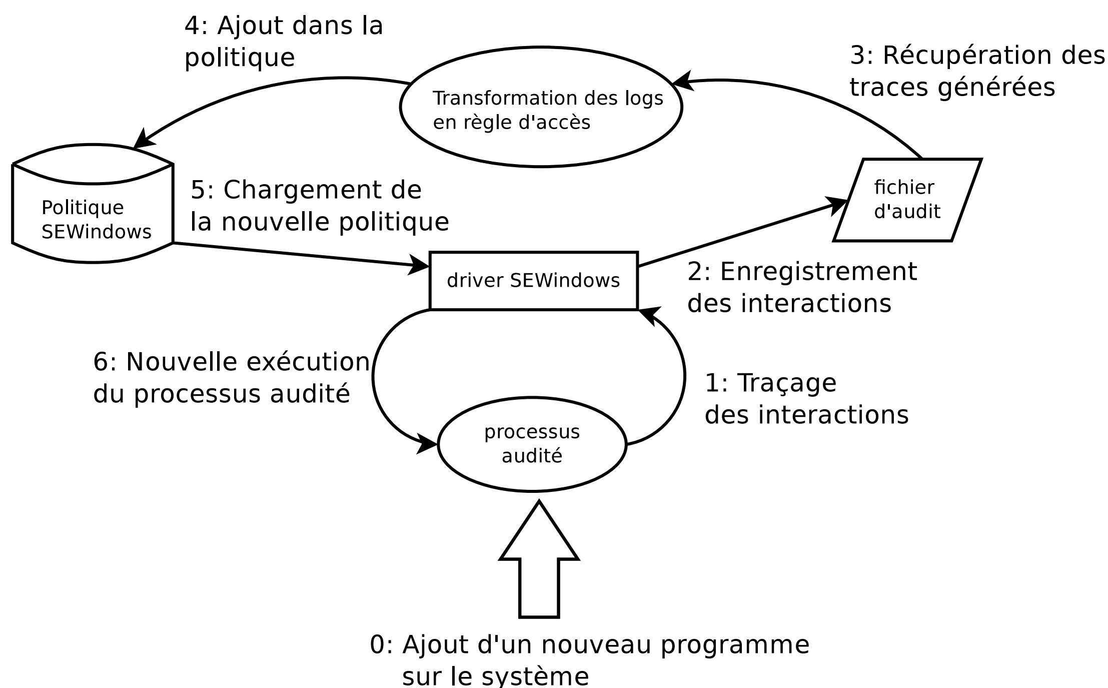

FIGURE 5.1 – Schéma décrivant l'apprentissage d'une politique

Avantages Le principal avantage de cette méthode d'écriture de la politique de contrôle d'accès est sa grande facilité de mise en place. En effet, il suffit de lancer le programme d'apprentissage et de réaliser les tâches nécessaires au sein du programme que l'on veut ajouter pour que les règles soient automatiquement générées. En plus de facilité l'écriture, cela l'accélère aussi. De plus, si l'administrateur a minutieusement réalisé toutes les tâches désirées au sein du nouveau programme, il ne devrait pas avoir à modifier plus tard sa politique. Cependant, lors d'une mise à jour du programme, il peut être nécessaire de recommencer l'opération pour ajouter automatiquement les nouvelles règles dans la politique mais les tâches sont prédéfinies.

Inconvénient Le principal inconvénient de cette méthode est la nécessité pour l'administrateur de vérifier, à la main, les règles qui ont été ajoutées par l'outil dans la politique. En effet, il est possible que le programme que l'on veuille ajouter, réalise des interactions malveillantes ou illégitimes pour l'administrateur, cachées par des tâches "normales". Par exemple, on pourrait imaginer un programme qui tente de copier la partie du registre contenant les mots de passe des utilisateurs. Lors de la phase d'apprentissage, cette interaction ne sera pas bloquée par le moniteur et par apprentissage, elle sera ajoutée à la politique courante. Si l'administrateur ne vérifie pas les règles, cette interaction malveillante sera aussi autorisée même en mode protection.

Il est donc nécessaire pour l'administrateur de vérifier les règles qui ont été ajoutées pour un programme. C'est pour cela que nous conseillons d'avoir au préalable une politique déjà écrite prenant en compte le reste du système, pour ne pas être parasité par les autres applications.

Manuelle

La seconde méthode pour créer une politique est de le faire de façon manuelle. Nous pouvons définir deux cas : soit l'administrateur connait précisément ce que fait le programme qu'il veut ajouter, dans ce cas, il écrit toutes les règles, soit il utilise la même méthode que pour l'automatisation, en transformant manuellement les traces en règles d'accès.

Dans les deux cas, le moniteur peut être utilisé soit en mode détection pour plus de facilité, soit en mode protection.

Avantages Le principal avantage de cette méthode est que l'administrateur maîtrise entièrement le processus de création des règles d'accès. Il peut ainsi vérifier qu'il n'autorise pas une interaction malveillante ou illégitime pour le nouveau programme.

Inconvénient Le principal inconvénient est que cette méthode est beaucoup plus coûteuse en temps. En effet, la retranscription à la main de chaque règle peut se révéler être un travail très fastidieux, surtout dans le cas des mises à jour des programmes qui les modifient intégralement, voire pour les mises à jour du système.

5.1.5 Discussion

Nous venons de détailler l'implantation choisie pour les modèles de protection PBAC et DTE appliqués aux systèmes Windows. Dans cette section, nous avons dû résoudre la problématique des ressources.

Pour le modèle de protection PBAC, nous avons choisi d'implanter le mécanisme de `namespace` en utilisant les variables d'environnement. Ces variables sont divisées en deux types : le premier est géré directement par le système. On le retrouve ainsi sur tous les systèmes Windows. Cela assure une portabilité complète de nos politiques de contrôle d'accès. Le second type est géré par l'administrateur. Il peut ainsi ajouter des variables d'environnement pour rédiger des règles

d'accès propres à son système ou à son parc de machines. Nous avons aussi ajouté à ce modèle de protection, un second modèle de protection nommé TPE. Il permet de définir des répertoires de confiance d'où seront lancés les fichiers exécutables.

En ce qui concerne le modèle de protection DTE, la principale difficulté était de trouver un moyen de stocker les contextes de sécurité des objets. Nous avons vu que SELinux utilisait les attributs étendus du système de fichiers pour les stocker. Cette solution n'est pas portable sur les systèmes Windows qui n'utilisent pas le même système de fichiers. Nous avons ainsi proposé une méthode évitant d'avoir à stocker les contextes de sécurité. Afin de ne pas créer de faiblesse dans notre modèle de protection, nous avons dû définir des contextes de sécurité plus précis que ceux proposés par SELinux. Pour cela, nous avons choisi de mettre dans le type du contexte de sécurité objet le chemin complet de l'objet. Cette méthode conduit à avoir les mêmes problèmes que pour le modèle PBAC, à savoir un problème dans le mécanisme de noms des ressources. Pour résoudre ce problème, nous avons utilisé la même solution que pour le modèle PBAC. Nous avons donc utilisé les variables d'environnement pour la définition des chemins complets des objets.

Grâce à l'implantation basée sur les variables d'environnement nous proposons donc des noms symboliques absolus indépendants de la localisation tels que définit dans le chapitre 3. Nous offrons donc un mécanisme permettant de désigner de manière précise chaque ressource du système. Nous avons montré que nous pouvions appliquer cette méthode non seulement sur les ressources du système de fichiers mais aussi sur les éléments du registre, mécanisme important dans les systèmes Windows.

Nous avons aussi détaillé les deux modes de fonctionnement de notre moniteur de référence. Le mode détection, qui va permettre d'écrire une politique de contrôle d'accès et un mode protection, qui l'applique strictement. Grâce au mode détection, nous avons mis en place un outil qui automatise la création de nouvelles règles pour l'ajout d'un nouveau programme. Cette automatisation n'est pas sans risque puisque l'outil n'effectue aucun contrôle sur la légitimité des règles ajoutées. Il est donc nécessaire que l'administrateur vérifie les règles générées.

Enfin, nous avons présenté notre mécanisme d'audit, qui génère des traces permettant de connaître les interactions qui se sont déroulées sur le système, mais qui sont aussi utilisées pour créer des règles de contrôle d'accès, que ce soit de façon automatique ou manuelle.

5.2 Implantation des mécanismes de contrôle d'accès obligatoire

Dans cette section, nous allons présenter les implantations de deux mécanismes de contrôle d'accès obligatoire pour les systèmes d'exploitation Windows, que nous avons appelé SEWINDOWS. Cette section, qui est divisée en deux parties, propose l'implantation sous deux formes différentes du *Policy Enforcement Point*. Ces deux méthodes ont été décrites dans la section 3.3.

La première implantation que nous proposons, utilise le détournement de la table des appels système de Windows. Cette première version nous a permis de valider les politiques que nous avons générées. La seconde implantation utilise les *filter-driver* pour détourner les flux d'exécution. Cette seconde version respecte les préconisations faites par Microsoft pour surveiller l'activité du système.

5.2.1 Implantation du mécanisme de contrôle d'accès basé sur la modification de la table des appels système

L'implantation de cette solution s'appuyant sur le modèle de protection DTE a fait l'objet d'une publication [Gros *et al.*, 2012]. L'application qui en a été faite, dont la définition des propriétés de sécurité fut détaillée dans [Blanc *et al.*, 2012].

5.2.1.1 Architecture fonctionnelle du mécanisme de contrôle d'accès

La figure 5.2 décrit l'architecture logicielle de notre solution de contrôle d'accès pour les systèmes Windows. Nous présentons ici les composants de cette architecture. Le premier composant est le *driver*. C'est le point central du mécanisme. Il est responsable de la prise de décision (*Policy Decision Point*). Cette implantation reprend toutes les phases que nous avons détaillées dans la partie 3.1.2.

Dans la phase de pré-décision, le moniteur est chargé de récupérer toutes les informations nécessaires à la prise de décision. Il est aussi en charge d'horodater et de numéroter la trace correspondante à l'interaction directe observée. C'est aussi dans cette phase que le moniteur doit mettre en place la méthode de représentation du système. Dans le cadre du modèle PBAC, il transforme les chemins des ressources en nom absolu indépendant de la localisation. Dans le cadre du modèle DTE, il doit labelliser les ressources.

Ensuite, le moniteur doit prendre une décision pour autoriser ou non l'interaction au regard de la politique de contrôle d'accès. La décision est ensuite ajoutée à la trace pour qu'elle puisse être générée.

Enfin, dans la phase de post-décision, soit le moniteur autorise la poursuite de l'interaction soit il la refuse. Dans tous les cas, il renvoie à la fin une réponse au processus ayant réalisé l'interaction.

Dans cette implantation, les points d'application de la politique (*Policy Enforcement Point*) sont réalisés par un module du *driver* : dans ce cas la, il s'agit de la modification de la table des appels système. La figure 5.2 illustre l'architecture divisée en trois blocs de cette implantation. Comme le schéma le montre, le moniteur est placé au centre du flux d'exécution pour pouvoir contrôler en prétraitement et en post-traitement les interactions réalisées sur le système.

Le premier bloc, nommé **Appel de fonction**, correspond à l'utilisation régulière d'un appel de fonction depuis l'espace utilisateur. Une application, qui réside en espace utilisateur, réalise une interaction sur le système. Cette interaction se traduit par l'exécution d'un appel système. Pour exécuter cet appel système, le système va chercher son adresse dans la table des appels système. C'est cette table qui a été modifiée par le *driver* SEWINDOWS pour détourner le flux d'exécution régulier.

Le deuxième bloc correspond à la partie SEWINDOWS. Une fois l'interaction détournée, le moniteur va prendre une décision dans le but d'autoriser ou non l'interaction courante. Cette décision se fait par le serveur de sécurité qui, dans le cas du modèle DTE, labellise les ressources. Dans le cas du modèle PBAC, les chemins sont transformés en nom symbolique absolu indépendant de la localisation. Cette partie est réalisée par un module spécifique du *driver*. Une fois la décision prise, la trace est transmise au *thread d'audit* qui enregistre la trace dans le fichier d'audit.

Le troisième bloc correspond à l'exécution normale, c'est-à-dire sans détournement opéré par le *driver* SEWINDOWS. Les fonctions de la table des appels système permettent d'interagir avec les éléments du système tels que le registre, le réseau ou encore le système de fichiers.

5.2.1.2 Détournement de la table des appels système

Nous allons dans cette partie décrire, d'un point de vue technique, la méthode de détournement du flux d'exécution en modifiant la table des appels système. Pour cela, nous allons dans un premier temps détailler le fonctionnement d'un flux d'exécution classique, puis dans un second temps, les modifications faites par le *driver*.

Exécution régulière La figure 5.3 illustre un flux d'exécution régulier. Un processus fait une interaction sur le système de fichiers comme la création d'un fichier. Pour cela, le processus utilise la fonction *CreateFile* de l'*API Win32*. Avant d'entrer en espace noyau, le flux d'exécution passe par la bibliothèque `ntdll` qui est chargée de faire correspondre la fonction *CreateFile* de l'*API*

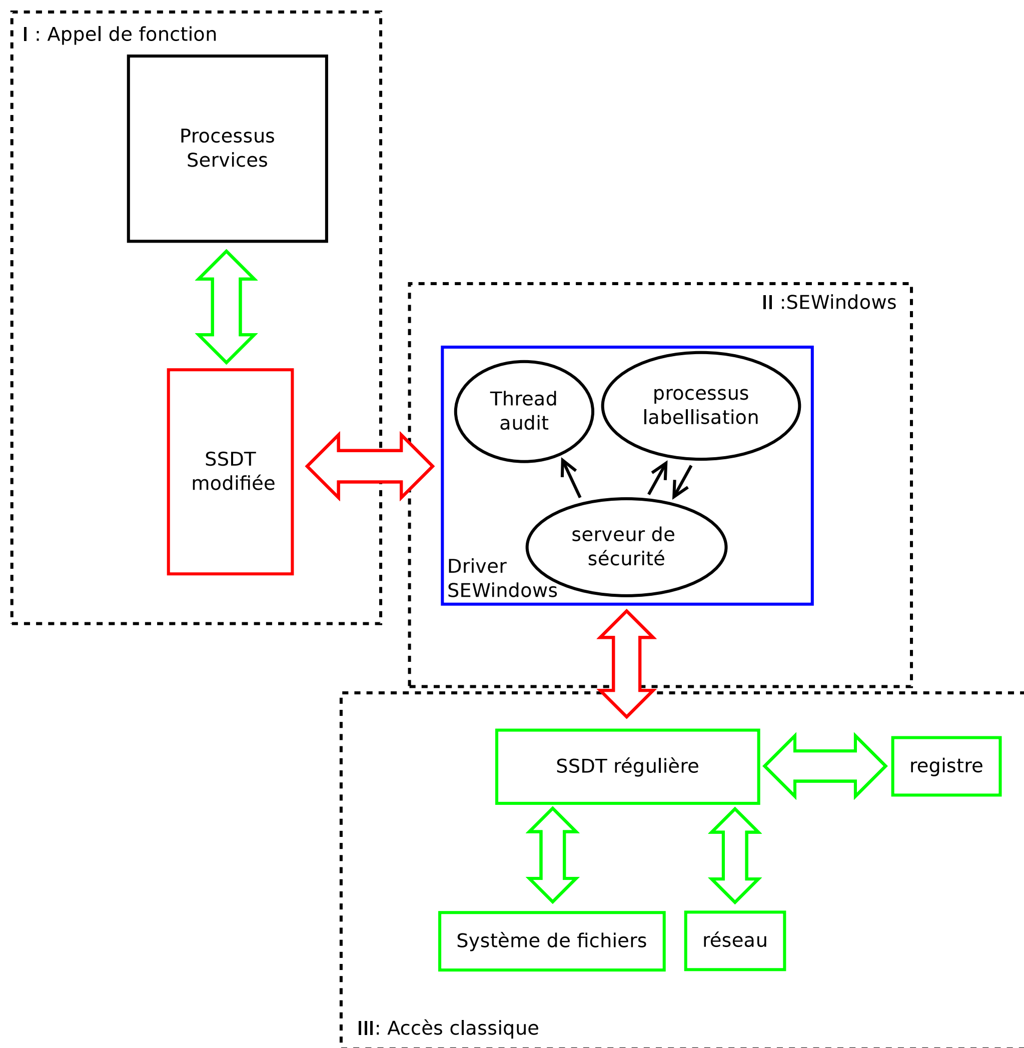

FIGURE 5.2 – Architecture globale

Win32 avec l'appel système correspondant qui est *NtCreateFile*. La bibliothèque `ntdll` ne passe au noyau que le numéro de l'appel système et non son nom. L'instruction processeur *SYSENTER* est enfin déclenchée pour effectuer le changement de contexte.

Une fois en espace noyau, le dispatcheur du noyau récupère le numéro de l'appel système et il lit dans la table des appels système SSDT l'adresse de la fonction à exécuter. Cette fonction se situe dans l'espace mémoire du noyau. Une fois l'appel système terminé, le retour de la fonction est renvoyé au processus ayant réalisé l'interaction. La vérification des droits d'accès est faite directement dans l'appel système.

Exécution modifiée Notre première implantation modifie le flux d'exécution régulier en altérant cette table des appels système. La figure 5.4 montre comment le détournement opère. Lorsque le *driver* SEWINDOWS se charge, il modifie la SSDT. Il remplace les adresses des appels système par des adresses de fonctions qu'il maîtrise.

À la différence des appels système dont les adresses sont situées dans l'espace mémoire du noyau, ces nouvelles adresses sont situées dans l'espace mémoire du *driver* SEWINDOWS. Le dispatcheur du noyau, qui lit l'adresse de l'appel système dans la SSDT obtient ainsi l'adresse

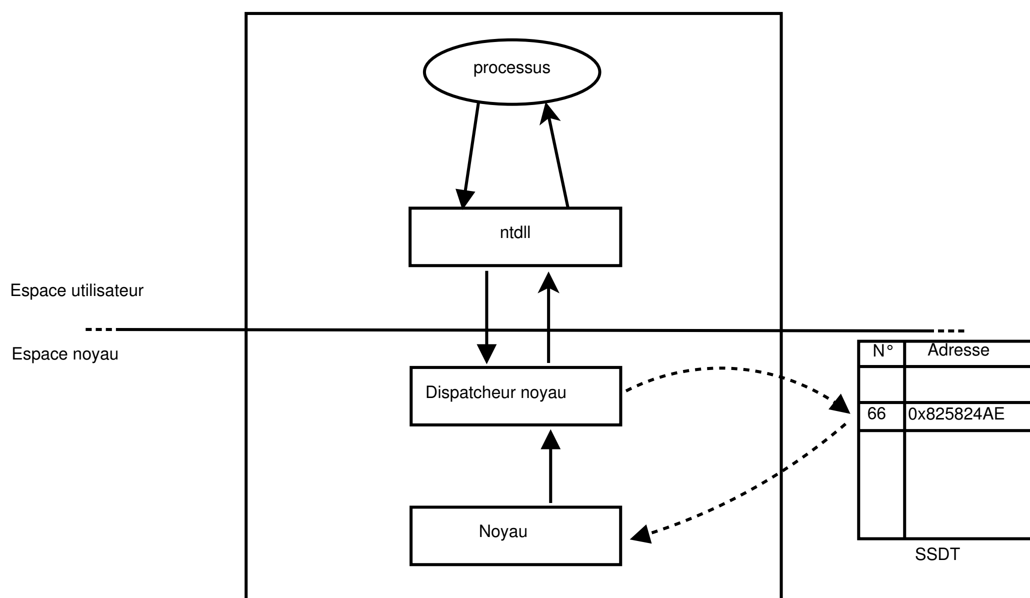

FIGURE 5.3 – Flux d'exécution classique

d'une fonction du *driver* SEWINDOWS. Le flux d'exécution est donc détourné dans le *driver* SEWINDOWS.

Le serveur de sécurité peut alors autoriser ou non l'exécution de l'appel système en fonction de la politique de contrôle d'accès qu'il a chargée. Toute interaction refusée par le serveur de sécurité est enregistrée dans le fichier d'audit.

L'altération de la table des appels système nous donne le contrôle des interactions ciblant le système de fichiers, des interactions sur le registre ainsi que des communications réseau. Les communications sont initialisées par l'utilisation de l'appel système *NtDeviceIoControl*.

Limites Cependant, la SSDT ne gère pas les appels système contrôlant la partie graphique de Windows. En effet, sur les systèmes d'exploitation Windows, la partie graphique est gérée indépendamment du système par le *driver win32.sys*. Elle possède, par conséquent, sa propre table d'appels système spécifique à la gestion graphique qui se nomme *Shadow SSDT*. Nous avons fait le choix de ne par modifier cette table système car son altération peut provoquer une instabilité du système.

En ne modifiant pas la *Shadow SSDT*, nous ne traitons pas les appels système qui gèrent la partie graphique de Windows. Un attaquant pourrait, s'il arrive à charger son *exploit*, réaliser des interactions que l'on peut considérer comme critiques. Par exemple, il serait capable de modifier ou voler les communications entre les applications. Il peut aussi forcer des applications à se fermer sans intervention de l'utilisateur.

Les tables des appels système ne sont accessibles et donc modifiables qu'en espace noyau, ce qui les protègent des attaques que pourraient réaliser des processus. Néanmoins, un attaquant qui arrive à charger un *driver* pourrait supprimer les modifications faites sur les SSDT sans que l'utilisateur ou l'administrateur ne s'en aperçoive. Il faut, pour détecter ces modifications, surveiller les modifications de ces tables. Ces limitations ont été traitées plus en détails dans la section 3.3.2.2.

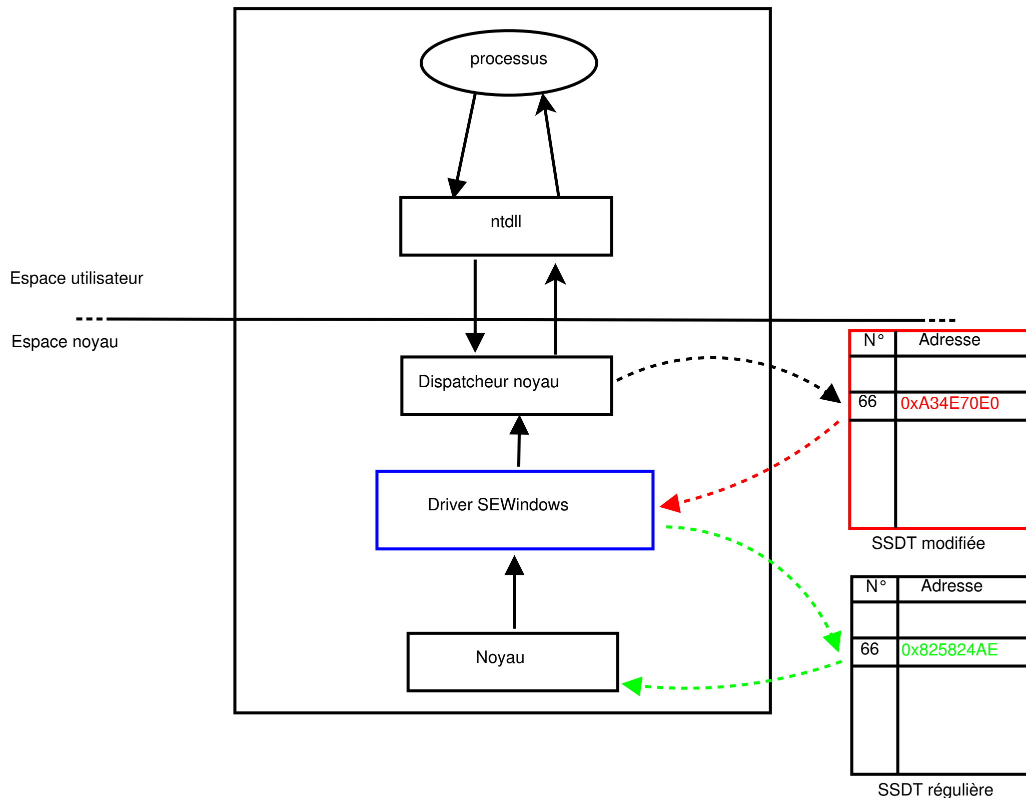

FIGURE 5.4 – Flux d'exécution détourné par la modification de la SSDT

5.2.2 Architecture basée sur les *filter-driver* et les *Kernel Callback*

Dans la première partie de cette section, nous avons mis en place une première architecture basée sur la modification de la table des appels système (SSDT). Cette solution, bien que fonctionnelle, n'est pas une solution pérenne.

En effet, cette première solution souffre de quelques limites qui peuvent se révéler très contraignantes. Tout d'abord, notre implantation n'est pas portable sur les Windows 64 bits puisqu'elle modifie une table du système qui est protégée par *Kernel Patch Protection*. Sur ces systèmes, *Kernel Patch Protection* vérifie l'intégrité des tables système et déclenche des exceptions lorsqu'elles sont modifiées. Ces exceptions conduisent à la création d'un *Blue Screen Of Death*. Il n'est donc pas possible de les modifier. Ensuite, la modification des tables système peuve rendre le système d'exploitation instable et peut générer des *Blue Screen Of Death*. Enfin, il est nécessaire d'avoir une certificat signé par Microsoft pour pouvoir charger un *driver* sur un système 64 bits.

Pour résoudre ce problème, il est nécessaire d'utiliser une méthode légitime pour détourner le flux d'exécution. Cette nouvelle solution modifie toujours le flux d'exécution du système sans pour autant altérer les tables système. Ce modèle est celui préconisé par Microsoft pour détourner le flux d'exécution. Il se base sur la mise en place de différents *driver* : les *filter-driver* et les *Kernel Callback*. Cette méthode a été décrite dans la partie C.3 (en Annexe).

5.2.2.1 Architecture globale

La figure 5.5 décrit la nouvelle architecture logicielle que nous avons mise en place. Cette nouvelle solution se base sur la création de trois nouveaux composants tout en conservant les éléments clés de l'architecture que nous avons décrite dans la partie 5.2.1. L'architecture de contrôle d'accès

globale n'est en rien modifiée. Le serveur de sécurité est toujours en charge d'appliquer strictement la politique de contrôle d'accès définie. Un processus de labellisation transforme les sujets et les objets pour leur associer des contextes de sécurité basés sur leur chemin complet. La partie traçabilité est conservée pour assurer des fichiers d'audit consistants détaillant avec précision les interactions passées sur le système.

Les modifications interviennent au niveau du détournement du flux d'exécution du système. Comme nous l'avons expliqué en introduction de cette partie, notre première implantation n'est pas pérenne par rapport à la politique de Microsoft mais surtout elle n'est pas portable sur les systèmes 64 bits. Les modifications introduites dans cette nouvelle architecture ont pour but de respecter les critères de développement et de surveillance édictés par Microsoft.

Nous avons donc une architecture de détournement des flux d'exécution du système divisée en trois composants principaux. Nous détaillerons plus la partie système que la partie réseau dans cette section. La raison est que la partie réseau est similaire dans la conception à la méthode utilisée par les *filter-driver* pour les entrées/sorties sur le système de fichiers.

Le premier composant de notre solution est un *filter-driver* qui gère les requêtes d'entrée/sortie sur le système de fichiers. De part son design spécifique, il n'est pas capable de gérer d'autres aspects du système, néanmoins, il offre la possibilité de pouvoir réaliser des traitements en prétraitement et post-traitement. Ce *filter-driver* est géré par un composant du système nommé le gestionnaire d'entrée/sortie auprès duquel il est nécessaire de s'enregistrer.

Le second composant est un *driver* filtrant les interactions sur le registre. Il est lui aussi capable de faire des contrôles en prétraitement et en post-traitement. Il s'enregistre auprès du gestionnaire d'entrée/sortie spécifique au registre.

Le troisième composant est un *driver* capable de contrôler le chargement et le déchargement de fichiers. Il peut ainsi contrôler le chargement des exécutables, mais aussi des bibliothèques ainsi que des *driver*. Il est aussi notifié de la mort d'un processus.

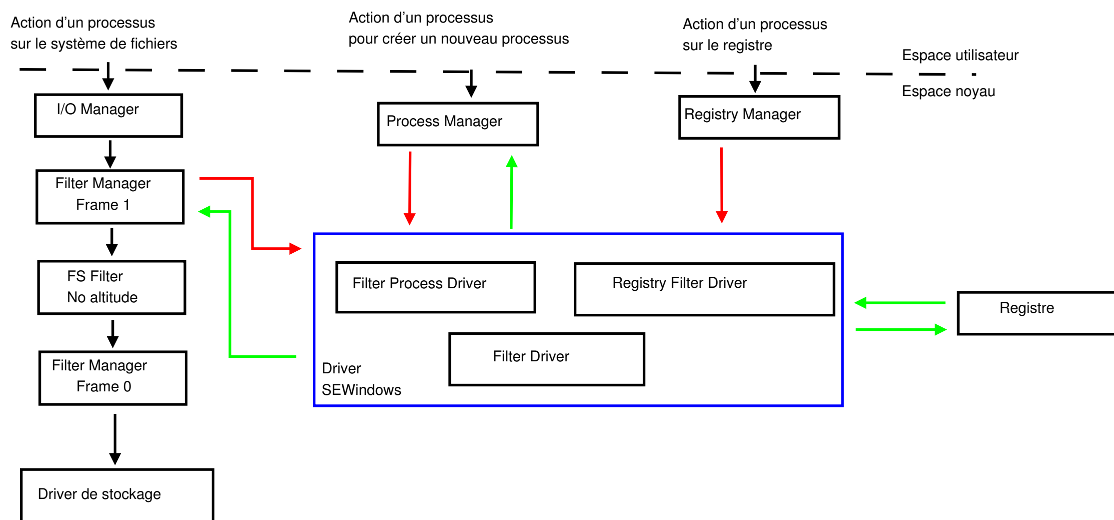

FIGURE 5.5 – Architecture de détournement basée sur l'utilisation de *filter-driver* et de *kernel callback*

5.2.2.2 Détournement des flux d'exécution

Détournement des interactions d'entrée/sortie Une limite du détournement des appels système par la modification de la SSDT est l'impossibilité de contrôler les interactions réalisées par les *driver*. En effet, la SSDT est essentiellement utilisée par les processus, qui résident en espace

utilisateur. Or, les *driver* n'utilisent pas les fonctions de la table des appels système pour réaliser des interactions sur le système.

Grâce à l'utilisation des *filter-driver* qui catégorisent les interactions sur le système de fichiers au lieu de détourner juste un appel système, il est aussi possible de contrôler les interactions faites par les *driver*. En effet, lorsqu'un *driver* utilise des fonctions comme *ZwCreateFile* ou *IoCreateFile*, le système convertit ces fonctions en IRP pour être traitées par les *filter-driver*.

La description de cette méthode a été faite dans la partie C.3. Nous allons juste reprendre quelques points importants.

Les *filter-driver* sont rangés grâce à un système d'altitude. Cette altitude détermine l'ordre de traitement des requêtes. Plus un *filter-driver* aura une altitude haute, c'est-à-dire proche du noyau, plus il recevra la requête tôt dans la phase de prétraitement. À l'inverse, dans la phase de post-traitement, il sera parmi les derniers à recevoir la requête. Donc la place choisie pour le *filter-driver* est très importante. Nous avons choisi comme altitude 360 000, qui est une altitude préconisée par Microsoft pour la catégorie de *filter-driver* que nous faisons.

Détournement des interactions sur le registre Les accès au registre passent par un gestionnaire d'entrée/sortie spécifique au registre. Pour contrôler les accès qui y sont fait, il est possible d'enregistrer un *driver* auprès de ce gestionnaire en utilisant le mécanisme de *Registry Callback*.

Grâce à cet enregistrement, le *driver* est capable de contrôler toutes les interactions sur le registre sans avoir à préciser les interactions qu'il désire traiter. Comme pour les *filter-driver*, le contrôle se fait en pré et post traitement.

Détournement des interactions de chargement et de déchargement des fichiers Les chargements des fichiers exécutables sont traités spécifiquement par le noyau Windows. Mais il est aussi possible pour un *driver* de contrôler le chargement de ces fichiers qui sont les plus dangereux pour le système puisqu'ils sont exécutés en espace noyau.

Le noyau Windows propose ainsi des routines de notification pour contrôler les créations de processus, le chargement de bibliothèques mais aussi de *driver*, appelées *Process Notify Routine*. Ces routines renseignent sur le nom du nouveau processus ainsi que sur son PID mais aussi sur le PPID. Il est ainsi possible de contrôler le chargement des fichiers exécutables.

Ces routines du noyau Windows renseignent aussi sur la mort d'un processus. Il n'est par contre pas possible de contrôler la mort d'un processus mais la notification offre la possibilité de l'inscrire dans un fichier d'audit.

Détournement des interactions sur le réseau Le détournement des interactions effectuées sur le réseau s'effectue par l'utilisation de la *Windows Filtering Platform*. Cette plateforme mise en place par Microsoft est similaire à celle des *filter-driver*. Grâce à l'utilisation des *API* que la plateforme fournie, il est possible de contrôler les interactions au niveau du réseau.

5.3 Expérimentations

Dans cette section, nous allons détailler les expérimentations réalisées grâce à notre moniteur pour Windows. Nous avons séparé cette section en deux parties distinctes. La première partie traite des expérimentations réalisées grâce à notre première implantation de SEWINDOWS basée sur le modèle DTE. Ces expérimentations nous ont permis de 1) générer des politiques ainsi que des `file_context` pour les systèmes d'exploitation Windows, 2) valider notre modèle de politique et 3) valider l'implantation que nous avons faite. Nous avons pour cela mis en place des scénarios d'attaque, basés dans un premier temps sur des interactions directes, c'est-à-dire des interactions

directes d'un sujet sur un objet, puis sur des scénarios complets avec des flux indirects en ajoutant PIGA à l'architecture de protection.

Dans une seconde partie, nous détaillons les expérimentations faites à partir de la seconde implantation de notre mécanisme de contrôle d'accès basée sur les *filter-driver*. Dans le but de générer des propriétés de sécurité et des configurations de mécanisme de contrôle d'accès pouvant contrer des menaces réelles, nous avons mis en place une architecture spécifique pour incorporer notre *filter-driver* au sein d'un mécanisme d'étude de logiciels malveillants (*sandbox*). En exécutant plusieurs logiciels malveillants et en étudiant les traces générées, nous allons écrire des règles de contrôle d'accès ainsi que des propriétés de sécurité pour PIGA afin de contrer explicitement ces comportements.

5.3.1 Violation d'une propriété de confidentialité

Dans cette partie, nous présentons les expérimentations réalisées grâce à notre mécanisme de contrôle d'accès. Nos expérimentations ne portent que sur l'implantation du modèle de protection DTE car nous voulons être en mesure d'exprimer des propriétés de sécurité grâce à PIGA.

Pour pouvoir exprimer ces propriétés de sécurité, PIGA utilise de façon préférentielle un fichier contenant les contextes de sécurité objets et d'une politique basée sur les types.

5.3.1.1 Environnement des expérimentations

Introduction Pour pouvoir réaliser nos expérimentations, nous avons dû définir une politique de contrôle d'accès autorisant le fonctionnement des programmes que nous voulions tester. Pour ce faire, nous avons utilisé le mode apprentissage de notre moniteur pour générer les politiques, c'est-à-dire que nous avions placé notre *driver* en mode évidence. Nous avons aussi utilisé la méthode automatique, c'est-à-dire que nous avons utilisé notre outil transformant les traces générés en règles de contrôle d'accès, tout en vérifiant qu'il n'y avait pas de règles illégitimes qui pourraient fausser nos expérimentations.

Plateforme Nous avons réalisé nos expérimentations dans une machine virtuelle avec un Windows 7 32 bits. Le *driver* SEWINDOWS était en mode apprentissage et nous avons fait des opérations légitimes sur la machine : exécution de `Windows Media Player`, `Firefox`, `Opera`, etc. Il est nécessaire de définir précisément les opérations que l'on souhaite autoriser par la suite. Cette planification se fait généralement à la main par l'utilisation d'une `check-list`.

Il faut bien évidemment refaire plusieurs fois les exécutions pour être sûr de n'avoir pas oublié de règles nécessaires au bon fonctionnement des applications. Une fois cette phase d'apprentissage terminée, nous disposions d'une politique suffisante pour que les applications puissent fonctionner correctement, même avec notre moniteur en mode protection.

La politique générée est suffisante pour un fonctionnement normal du système d'exploitation, c'est-à-dire que nous sommes capables de réaliser certaines tâches définies sans que le *driver* SEWINDOWS ne les bloque. Nous obtenons donc une politique SEWINDOWS avec environ 5000 règles d'accès et 40 000 contextes de sécurité.

La table 5.6 récapitule les informations sur la politique.

Nombre de contextes de sécurité objets	40 000
Nombre de contextes de sécurité sujets	1 800
Nombre d'interactions	5 000

FIGURE 5.6 – Statistiques sur la politique de contrôle d'accès pour le premier scénario

5.3.1.2 Scénario des attaques

Nous détaillons deux scénarios d'attaque pour montrer l'efficacité de notre solution. Le premier propose une attaque "simple" qui repose sur un scénario joué volontairement par l'utilisateur. Il essaye de contourner la politique de contrôle d'accès par une interaction directe. Le second scénario est plus réaliste puisqu'il utilise des flux indirects autorisés par SEWINDOWS.

Nous montrons dans le premier cas, qui correspond à un flux d'information direct, que notre moniteur est capable de l'interdire. A contrario, nous montrerons, dans le second cas qui résultat d'un flux d'information indirect, que notre moniteur, même s'il est capable de voir l'ensemble des interactions directes, n'est pas capable de bloquer la violation de propriété. Il est donc nécessaire d'utiliser un second moniteur pour détecter ce genre de violation.

5.3.1.3 Violation directe

Dans notre premier exemple, nous avons un utilisateur qui tente de lire le contenu d'un dossier auquel il n'a pas accès en passant par un navigateur Internet. Ce refus se traduit par la trace suivante listée en listing 5.17. Elle montre que le sujet ayant pour contexte de sécurité `system_u:system_r:firefox_t` veut lire (`read`) et traverser (`execute`) le dossier ayant pour contexte de sécurité `system_u:object_r:systemdrive|users|ensib|appdata|roaming|opera|opera_dir_t`

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

Listing 5.17 – Trace d'interdiction pour `Firefox` de lire le contenu du dossier d'`Opera`

5.3.1.4 Violation indirecte

Ce second scénario propose de reprendre l'attaque qui vise à enfreindre la confidentialité que l'on veut imposer entre nos deux navigateurs `Firefox` et `Opera`. Dans cette attaque, nous allons essayer de contourner la politique de contrôle d'accès en utilisant une suite d'interactions légitimes au système vis-à-vis de SEWINDOWS.

Via des propriétés de sécurité comme la confidentialité, PIGA est capable de détecter des violations indirectes de la confidentialité.

Le listing 5.18 montre la définition d'une propriété de confidentialité entre deux contextes de sécurité. Cette propriété spécifie qu'il ne doit y avoir aucun flux d'information entre le contexte de sécurité objet et le contexte de sécurité sujet. C'est pourquoi, dans notre propriété de confidentialité, nous interdisons les flux d'information indirects représentés par le symbole $>>$, allant de sc_2 vers sc_1.

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

Listing 5.18 – Définition de la propriété de confidentialité

Nous l'appliquons entre les navigateurs `Firefox` et `Opera`. Pour contrôler les flux entre les deux navigateurs, il est nécessaire d'écrire les deux propriétés de confidentialité comme le montre le listing 5.19.

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

Listing 5.19 – Application de la propriété de confidentialité

En rejouant des traces générées par notre *driver* SEWINDOWS dans PIGA et en appliquant la propriété de sécurité définie au listing 5.19, nous détectons un flux d'information entre `Firefox` et `Opera`. Ce transfert d'informations réussit grâce à l'utilisation d'un logiciel tiers, ici `Adobe`.

Le listing 5.20 montre une détection faite par PIGA. Nous avons `Firefox` qui écrit un fichier à la racine du système. Puis `Adobe` qui lit le contenu du fichier pour aller ensuite écrire un nouveau fichier dans le répertoire d'`Opera` qui va finir par le lire.

```
1 system_u:system_r:firefox_t -( file { create write } )->system_u:object_r:
      systemroot_file_t ;
2 system_u:system_r:adobe_t -( file { execute read } )-> system_u:object_r:
      systemroot_file_t;
3 system_u:system_r:adobe_t -( dir { create setattr unlink } )-> user_u:object_r:
      systemdrive|user|bob|AppData|Local|opera_dir_t ;
4 system_u:system_r:opera_t -( file { execute getattr read } )-> system_u:object_r:
      systemdrive|user|bob|AppData|Local|opera_file_t;
```

Listing 5.20 – Trace d'un flux indirect de Firefox vers Opera

La détection faite par PIGA pouvant être difficile à lire, nous proposons une illustration 5.7 qui montre le cheminement de l'attaque. `Firefox` écrit dans un fichier (flèche 1). Ce fichier est lu par `Adobe` (flèche 2). `Adode` écrit dans un second fichier, qui est dans le répertoire visé par l'attaque (*i.e* : le répertoire d'`Opera`). Enfin `Opera` lit le contenu de ce fichier.

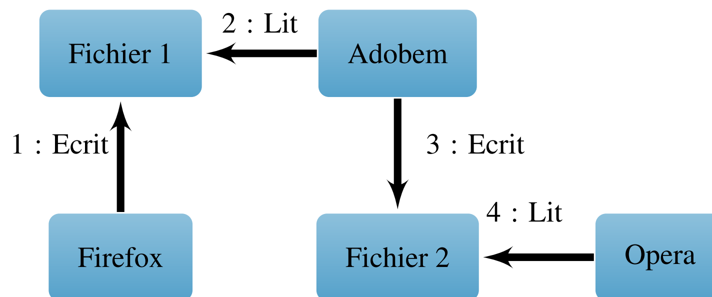

FIGURE 5.7 – Schéma du flux indirect conduisant à la violation de la propriété de confidentialité

Nous obtenons donc un flux d'information indirect entre les deux navigateurs, obtenu grâce à un programme intermédiaire, ici `Adobe Reader`.

5.3.2 Étude de logiciels malveillants

5.3.2.1 Introduction

Dans cette partie, nous développons les tests que nous avons réalisés sur une architecture spécifique que nous avons mise en place. Ces tests se sont orientés sur l'analyse de logiciels malveillants. Dans cette configuration, nous avons dû mettre en place une solution pour assurer l'intégrité des fichiers d'audit que nous récupérons lors de l'exécution du logiciel malveillant. Cette solution consiste à envoyer les traces générées par le moniteur à un serveur de sécurité.

Les expérimentations que nous avons faites ont été réalisées en machine virtuelle et nous décrivons dans cette partie les protocoles qui ont été appliqués sur deux types de logiciels malveillants. Nous avons mis en place des outils facilitant la lecture des traces générées dans le but de comprendre l'impact du logiciel malveillant sur le système et pour connaître les fichiers contaminés.

Ces outils offrent une représentation textuelle et graphique des résultats ce qui facilite la compréhension des mécanismes d'installation du logiciel malveillant.

En plus de ces outils de visualisation, nous proposons des méthodes pour dériver une politique de contrôle d'accès pour SEWINDOWS à partir des traces générées. Ces politiques empêchent l'installation de ce type de logiciel malveillant en traitant les interactions directes entre les sujets et les objets du système. Cependant, comme ces politiques empêchent les interactions illégitimes, nous proposons aussi des outils pour en dériver des propriétés de sécurité qui pourront être appliquées par PIGA.

Exécution d'un logiciel malveillant

Ce premier scénario propose l'exécution d'un logiciel malveillant plutôt basique. En utilisant le mode détection de notre mécanisme, nous montrerons les comportements clés du logiciel malveillant. Nous avons commencé par étudier un *FakeAV* dont l'analyse par VirusTotal se trouve ici [1].

Comme le binaire n'est pas présent dans la politique nous avons mis notre *driver* SEWINDOWS en mode détection pour dérouler l'installation du logiciel malveillant et nous montrerons des points clés dans son installation que notre mécanisme de contrôle d'accès est capable de bloquer.

Le premier point important est le moment où le logiciel malveillant est exécuté. Cela se traduit par le chargement (*load*) du binaire par `explorer.exe`. Cette trace est retranscrite par notre *driver* SEWINDOWS dans le listing 5.21. Nous avons volontairement renommé le logiciel malveillant en `malware` pour pouvoir plus facilement suivre son évolution dans les traces. Le logiciel malveillant est exécuté par nos soins.

```
1 type=DTE audit(129676508858814710,43) avc:denied { execute } for pid=308
2 com="%systemroot%\explorer.exe" ppid=360
3 path="%systemdrive%\users\bob\desktop\malware.exe"
4 scontext=system_u:system_r:explorer_t
5 tcontext=system_u:object_r:systemdrive|users|bob|desktop|malware_exec_t
6 tclass=load
```

Listing 5.21 – Chargement du logiciel malveillant par `explorer.exe`

Ensuite, le logiciel malveillant va exécuter un invite de commande comme le montre le listing 5.22.

```
1 type=DTE audit(129676508874908460,159) avc:denied { execute } for pid=3376
2 com="%systemdrive%\users\bob\desktop\malware.exe" ppid=308
3 path="%systemroot%\system32\cmd.exe" scontext=system_u:system_r:malware_t
4 tcontext=system_u:object_r:systemroot|system32|cmd_exec_t tclass=load
```

Listing 5.22 – Exécution du premier invite de commande par le logiciel malveillant

Ce premier invite de commande exécute un second invite de commande et écrit un nouveau fichier qui se nomme `script.bat`. 5.23.

```
1 type=DTE audit(129676508874908460,171) avc:denied { execute } for
2 pid=2628 com="%systemroot%\system32\cmd.exe" ppid=3376
3 path="%systemroot%\system32\cmd.exe" scontext=system_u:system_r:cmd_t
4 tcontext=system_u:object_r:systemroot|system32|cmd_exec_t tclass=load
5
6 type=DTE audit(129676508875064710,177) avc:denied { write } for pid=3376
7 com="%systemdrive%\users\bob\desktop\malware.exe" ppid=308
8 path="%systemdrive%\users\bob\desktop\script.bat"
```

1. `https://www.virustotal.com/en/file/d0d0d53f66b400cb43b3019be9e5e49a9097458119eba8f18d27d9e7b4` `analysis/`

```
9  scontext=system_u:system_r:malware_t
10 tcontext=system_u:object_r:systemdrive|users|bob|desktop|script_bat_t
11 tclass=file
```

Listing 5.23 – Exécution d'un second invite de commande et écriture du script

Enfin, le second invite de commande va lire le contenu du script pour exécuter les commandes qui sont à l'intérieur. 5.24.

```
1 type=DTE audit(129676508875689710,221) avc:denied { read } for pid=2832
2 com="%systemroot%\system32\cmd.exe" ppid=2628
3 path="%systemdrive%\users\bob\desktop\script.bat"
4 scontext=system_u:system_r:cmd_t
5 tcontext=system_u:object_r:systemdrive|users|bob|desktop|script_bat_t
6 tclass=file
```

Listing 5.24 – Lecture du script par le second invite de commande

Grâce à la précision des fichiers d'audit, nous pouvons facilement suivre l'évolution de ce logiciel malveillant sur le système. Il nous est facile de suivre les relations de filiation grâce notamment aux PID et aux PPID.

À travers l'étude des traces générées par notre moniteur, nous avons pu suivre l'évolution de l'installation du logiciel malveillant. Nous ne montrons pas tous les éléments de cette infection, mais nous pouvons bloquer une par une les interactions réalisées par le logiciel.

Ce scénario, entièrement joué puisque le moniteur est en mode détection, est un cas concret d'exécution du logiciel malveillant par un navigateur soumis à une attaque.

5.3.2.2 Architecture décentralisée

À partir de l'implantation du mécanisme de contrôle d'accès basée sur les *filter-driver* et les *kernel callback*, nous avons mis en place une architecture spécifique permettant d'analyser les logiciels malveillants. L'objectif de ces expérimentations est d'exécuter les logiciels malveillants sans bloquer leurs interactions dans le but de connaître leur impact sur le système entier et d'en étudier leur comportement.

Comme notre *driver* est en mode détection pour ces études, il enregistre chaque interaction effectuée sur le système. De plus, pour être sûr de ne pas rater d'interaction, il n'y a pas de politique de contrôle d'accès. Cependant, le logiciel malveillant pourrait modifier le contenu du fichier d'audit, voire le supprimer puisqu'il est stocké sur le système et que les interactions ne sont pas bloquées par le moniteur. Pour éviter d'avoir des fichiers d'audit potentiellement modifiés par le logiciel malveillant, nous avons décidé d'envoyer directement les traces sur un serveur de sécurité distant.

Instrumentation du système La figure 5.8 représente notre nouvelle architecture du côté du Windows instrumenté. Cette architecture est divisée en quatre parties distinctes.

Le premier bloc traite les accès au système de fichiers. Lorsqu'une interaction est réalisée sur le système de fichiers, l'interaction est interceptée lors de son prétraitement par notre *filter-driver* (flèche 1, 2, 3). Le *filter-driver* est chargé de construire une trace suffisamment précise pour pouvoir avoir toutes les informations nécessaires à l'étude du logiciel. De plus, pour avoir une chronologie des différentes interactions réalisées, nous ajoutons un identifiant unique pour chaque interaction ainsi qu'un *timestamp*. Nous obtenons ainsi une trace contenant :

1. un numéro de trace unique, ainsi que le numéro du cœur processeur qui exécute l'action ;
2. un `timestamp` ;

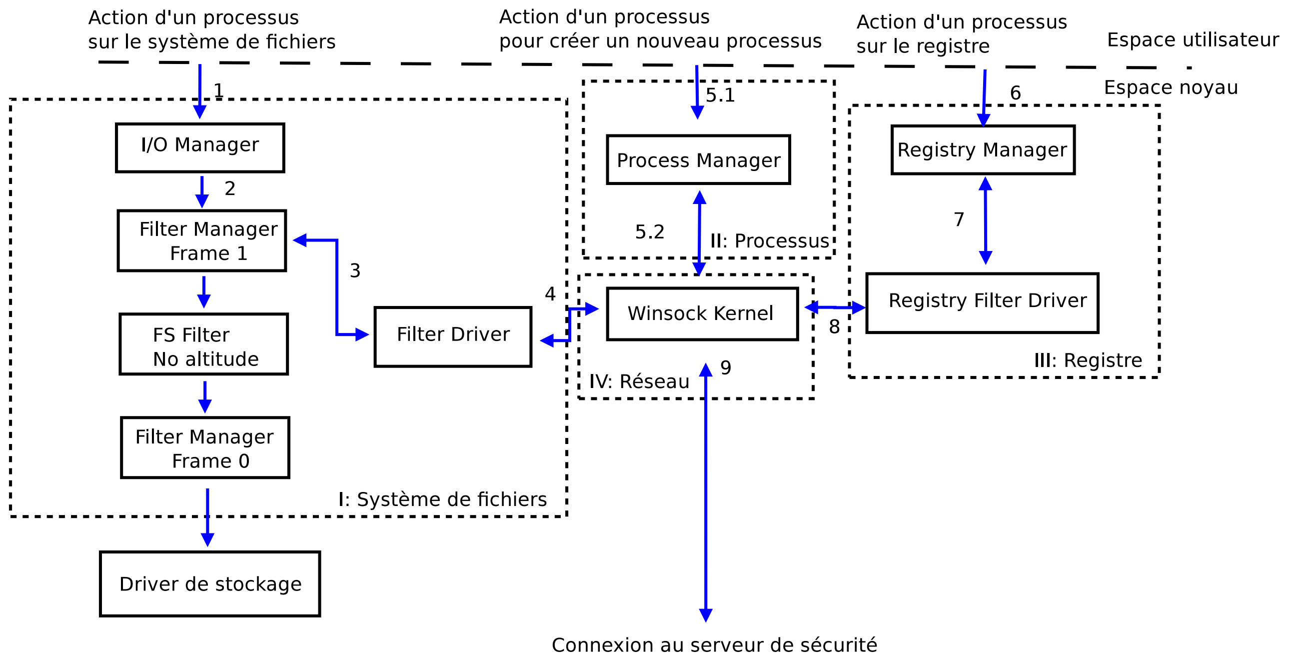

FIGURE 5.8 – Instrumentation du système pour l'analyse de logiciel malveillant

3. le type d'accès ;
4. nom court du processus qui a fait l'action et PID ;
5. nom de l'objet.

Dans le but de ne pas faire de calcul intempestif sur la machine d'étude, nous ne calculons pas de contexte de sécurité. Cette tâche sera faite par le serveur de sécurité dans un mode *offline*. Un exemple de trace générée est illustré par le listing 5.25. Cette trace est ensuite envoyée (flèche 4) au *thread* qui s'occupe de la communication avec le serveur de sécurité.

```
1 trace=1118 , cpu=0
2 timestamp=130208621983766250
3 Access : create
4 pid : consent.exe 568
5 Object Name : \Device\HarddiskVolume2\Users\toto\AppData\LocalLow
```

Listing 5.25 – Trace générée par le *filter-driver* pour une interaction sur le système de fichiers

Le second bloc traite de la gestion de processus. À chaque création ou mort d'un processus, un *thread* du *filter-driver* enregistre certaines informations. Il doit récupérer les noms complets des processus impliqués, c'est-à-dire du père et du fils, ainsi que leur PID (flèche 5.1 et flèche 5.2). Une fois que ces informations sont enregistrées, la trace générée pour la création d'un processus 5.26 est envoyée au *thread* qui s'occupe de la communication avec le serveur de sécurité. La même chose est faite pour la mort d'un processus 5.27. Nous ne datons pas ces traces car ce n'est pas nécessaire. Nous nous servons de ces traces pour connaître les chemins complets ainsi que les PID des processus créés et détruits.

```
1 Parent Process name : \Device\HarddiskVolume2\Windows\System32\svchost.exe pid
    =908
2 Process name : \Device\HarddiskVolume2\Windows\System32\consent.exe pid=568
    ppid=908 sid=1
```

Listing 5.26 – Trace générée par le *filter-driver* pour la création d'un processus

```
1 Parent Process name : \Device\HarddiskVolume2\Windows\System32\svchost.exe pid
      =908
2 Process name : \Device\HarddiskVolume2\Windows\System32\consent.exe pid=568
      ppid=908 sid=1
```

Listing 5.27 – Trace générée par le *filter-driver* pour la mort d'un processus

Le troisième bloc de l'architecture concerne la gestion du registre. Tout comme pour le système de fichiers, nous n'interceptons que les requêtes en prétraitement. Lorsqu'un processus fait une interaction sur le registre, elle est interceptée (flèche 6) par le gestionnaire de registre puis transmise (flèche 7) à notre *driver*. Le *Registry Filter Driver* récupère toutes les informations nécessaires à la compréhension de l'interaction. Il ajoute aussi un identifiant unique ainsi qu'un *timestamp*. La trace construite est très proche de la trace générée pour le système de fichiers. Mais, en plus des informations de type nom du processus ainsi que son PID et nom de l'objet, qui dans ce cas la correspond à une clé registre, nous ajoutons le couple valeur et donnée dans la trace, pour connaître précisément les interactions du logiciel étudié. Nous obtenons ainsi une trace complète, comme le montre le listing 5.28, qui est envoyée au *thread* qui s'occupe de la communication avec le serveur de sécurité.

```
1 trace=2258 , cpu=0
2 timestamp=130257368578773750
3 Access : set_value_key
4 pid : explorer.exe 1644
5 \REGISTRY\USER\S-1-5-21-2571257828-2288546103-543878076-1001\Software\Microsoft
      \Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-ACE2-4F4F-9178-9926
      F41749EA}\Count
6 {Q65231O0-O2S1-4857-N4PR-N8R7P6RN7Q27}\pzq.rkr REG_BINARY :
      0000000025000000080010000095CA5300000080BF000080BF000080BF000080BF
7 000080
      BF000080BF000080BF000080BF000080BF000080BFFFFFFFFFC0B06AEB62C4CE0100000000
```

Listing 5.28 – Trace générée par le *driver* pour une interaction sur le registre

Enfin, le quatrième bloc est celui qui gère les communications avec le serveur de sécurité. L'implantation choisie sur le système instrumenté doit assurer deux règles importantes : la première règle est que les communications avec le serveur de sécurité distant doivent être assurées par un module dédié de l'architecture pour ne pas interférer avec le reste du mécanisme. Pour cela, nous utilisons un *thread* dédié qui ne s'occupera que des communications réseau. La seconde règle impose que cette communication ne doit ni ralentir, ni bloquer le fonctionnement de la machine.

Pour cela, nous avons choisi une implantation basée sur le principe de producteur/consommateur. Les producteurs sont les trois parties précédemment présentées. Elles approvisionnent une file de messages sans attendre que le message soit envoyé. Ainsi, elles n'ont pas besoin d'attendre un retour de la part du *thread* qui envoie les traces et elles ne bloquent donc pas les interactions. Le consommateur est par conséquent le *thread* qui récupère les traces et les envoie au serveur de sécurité.

Les communications sont réalisées sur une connexion TCP.

Nous noterons une chose importante. Nous avons ajouté dans les traces traitant les interactions du système de fichiers et du registre, un identifiant unique ainsi qu'un *timestamp*. Il est indispensable pour la compréhension de l'installation de l'infection que cet identifiant soit unique pour tout le système. Il est donc partagé entre les différentes parties de notre architecture.

De plus, comme nous le montrons dans les traces, nous n'enregistrons que le nom court des processus. Ce nom est récupéré grâce à la fonction `ZwQueryInformationProcess`. Elle va lire un champ de la structure `EPROCESS` [2].

Une étape importante est donc réalisée au moment du démarrage du *driver*. Lors de cette étape, il va établir la liste des processus qui sont en cours de fonctionnement en recherchant, quand c'est possible, le père de chaque processus. En effet, sous Windows, il est possible qu'un processus n'ait pas de père (à la différence des systèmes Linux où les processus sont toujours au moins rattachés à `init`). Nous obtenons ainsi un graphe de l'activité courante du système. Cela nous permet aussi de connaître les chemins complets ainsi que les PID des processus qui sont lancés au moment où nous exécutons le logiciel étudié.

Serveur de sécurité distant Le serveur de sécurité est divisé en deux parties : la première partie est un serveur TCP, fait en python, qui récupère les traces envoyées par le *driver* pour les mettre dans une base de données. Comme le nombre de traces est relativement grand, la mise en base est faite de manière *offline* pour ne pas surcharger la machine. La seconde partie permet la visualisation des résultats. Elle propose la génération de rapport présentant les activités qui se sont déroulées sur le système étudié.

5.3.2.3 Protocoles des expérimentations

Plateforme Nous avons fait le choix d'instrumenter un Windows 7 32 bits. Même si cette nouvelle implantation nous permet de tester sur les architectures 64 bits, il est nécessaire de signer le *driver* avec un certificat émis par une autorité reconnue par Microsoft pour que le *Kernel Patch Protection* l'autorise à se charger. Il existe cependant un mode appelé *testsigning* qui permet d'auto-signer son *driver* avec un certificat que nous aurions créé. Néanmoins, dans ce mode, *Kernel Patch Protection* ne vérifie plus que la validité de la signature, ce qui modifie le comportement du système. Si un logiciel malveillant charge un *driver* auto-signé, il ne sera pas bloqué par les protections natives du système.

Ces tests ont été réalisés dans une machine virtuelle pour bénéficier de la possibilité de faire des instantanés des machines et ainsi éviter de les réinstaller entre chaque étude. De plus, cela nous permet de connaître parfaitement l'état de départ de la machine qui sert pour les tests.

Le serveur de sécurité est une machine capable de faire fonctionner un serveur web ainsi que des scripts pythons, dans notre cas, un système Linux.

Exécution des logiciels malveillants Les logiciels malveillants étudiés sont téléchargés directement sur la machine.

Les logiciels malveillants ont besoin d'être exécutés *à la main* car il faut pouvoir autoriser son exécution si une demande d'élévation des privilèges est demandée. Cette manipulation nous permet à la fois de connaître le nom du binaire qui est exécuté mais aussi de savoir comment il doit être lancé. Nous savons par cette manipulation que le processus, qui lance le logiciel malveillant, est `explorer.exe`. Cette information facilite le traitement des résultats.

Une fois que le logiciel malveillant est installé, nous laissons l'infection se développer sur le système. Il est parfois nécessaire que valider des opérations intermédiaires à cause de l'*User Access Control*. Nous acceptons à chaque fois les demandes faites par l'*User Access Control*.

Nous avons fait le choix de ne pas désactiver l'*User Access Control* car cette désactivation fausserait aussi l'étude. En effet, lorsque cette protection n'est plus activée, des mécanismes de

2. `EPROCESS` est une structure du noyau de Windows contenant les informations sur les processus (nom, PID, *thread*, etc). Cette structure est opaque, c'est-à-dire qu'il est nécessaire d'utiliser une fonction pour récupérer des informations dans les champs de structure

sécurité ne sont plus appliqués. Par exemple, il n'y a plus de vérification des niveaux d'intégrité lorsqu'un sujet effectue une interaction sur un objet.

Serveur de sécurité Une fois que l'utilisateur considère que l'exécution est terminée, le serveur de sécurité met dans une base de données les informations recueillies par le *driver* pour les conserver et les comparer aux autres études. Il génère ensuite les résultats pour qu'ils puissent être analysés.

5.3.2.4 Visualisation des résultats et définition des propriétés de sécurité

Une fois le logiciel malveillant exécuté, il faut pouvoir fournir des résultats pertinents, c'est-à-dire des résultats qui permettent de comprendre le cheminement de l'infection : nous ciblerons donc les interactions spécifiques au logiciel malveillant qui permettent la propagation de l'infection au sein du système.

Pour ce faire, nous proposons un outil graphique générant les résultats sous la forme d'un graphe. Ce graphe est une représentation du système ayant pour nœud les processus et comme feuilles les interactions réalisées ainsi que les objets. Nous proposons aussi un outil qui extrait les interactions directes sous la forme d'automates. Ces automates sont des sous graphes, qui représentent une partie du comportement du logiciel malveillant étudié. À partir de ce sous graphe, il nous est possible d'écrire des règles de contrôles d'accès empêchant le déploiement de l'infection, mais aussi des propriétés de sécurité qui peuvent être vérifiées et appliquées par PIGA.

Visualisation

Pour comprendre le cheminement utilisé par l'infection, nous présentons les résultats sous une forme brute. Dans une première partie, nous listons les processus ainsi que leurs parents.

Associées à ces processus, il est possible de connaître toutes les interactions directes générées sur le système par ce processus sans aucun filtre. Cette première visualisation permet de connaître l'activité complète du processus (fichier créé, supprimé, processus lancé, clé registre modifiée, etc.). Cette forme est une forme brute des traces générées, triées par ordre temporaire.

Dans le but de mieux comprendre les interactions passées sur la machine d'analyse, il est nécessaire de travailler les résultats. La première chose est de visualiser quelques interactions précises.

Nous avons choisi d'isoler pour la partie système de fichiers :

- les chargements,
- écriture et suppression.

Ce choix est motivé par le fait que nous voulons suivre la propagation de l'infection au sein du système et ainsi connaître les éléments qui ont été infectés. Donc nous isolons les interactions qui permettent d'échanger de l'information depuis un processus infecté vers un objet du système.

Ce modèle n'est pas figé. En rajoutant par exemple, les opérations élémentaires de lecture, nous pouvons détecter les fuites d'information ou les violations de confidentialité.

Nous allons détailler nos résultats sur l'analyse d'un logiciel malveillant appelé *ZeroAccess*. Ce `malware` est celui que nous avons utilisé dans notre introduction 1.3. Nous allons chercher à comprendre son comportement lors de son installation pour en extraire les règles de contrôle d'accès nécessaires pour empêcher sa propagation au sein du système.

Génération du graphe des processus La représentation sous la forme d'un graphe nous permet de connaître quelques interactions spécifiques. La figure 5.9 montre le premier résultat obtenu. À partir du fichier de l'infection nommé `xxx-porn-movie.avi.exe`, nous remarquons qu'il va

écrire des fichiers dans la corbeille de Windows qui se nomme `Recycler.Bin`. On peut tout d'abord noter que le logiciel malveillant veut se cacher en utilisant une double extension, pour tromper l'utilisateur en lui faisant croire que c'est une vidéo dans le but qu'il exécute le fichier binaire.

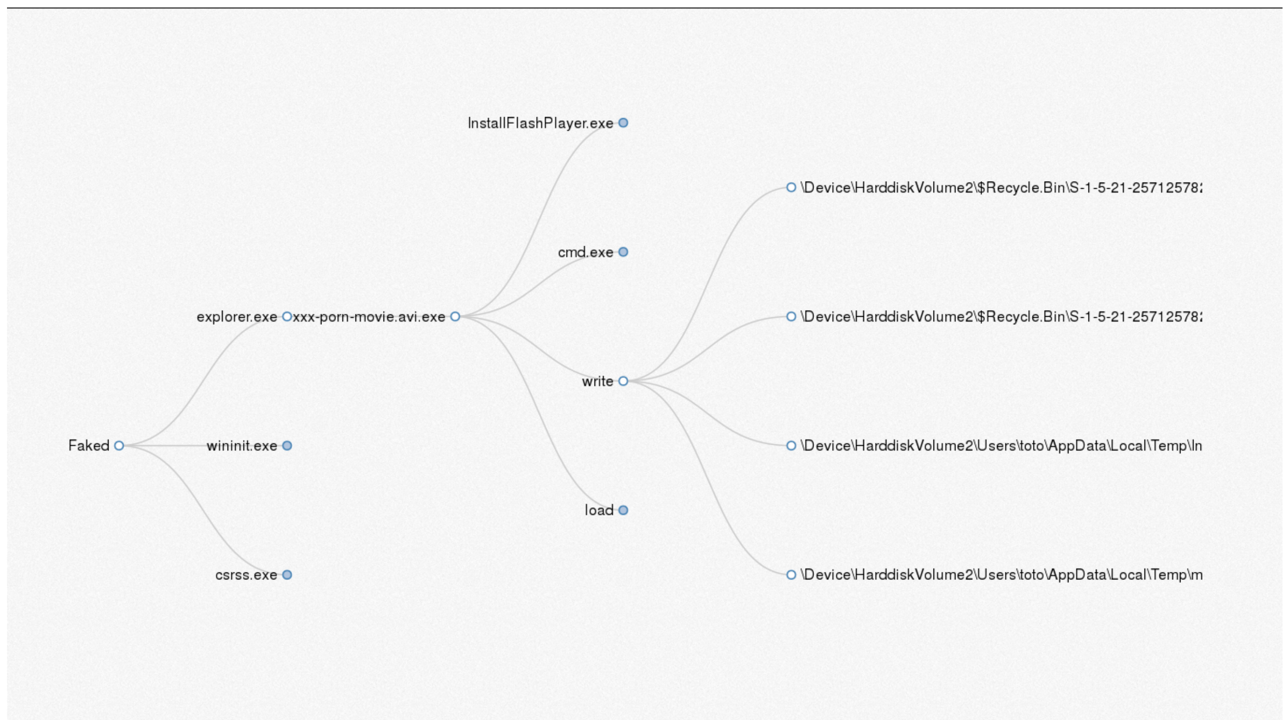

FIGURE 5.9 – Logiciel malveillant écrivant des fichiers dans la corbeille de Windows

Le graphe nous montre un lien entre notre fichier principal de l'infection et un processus qui semble légitime nommé `InstallFlashPlayer.exe`. Nous pouvons voir que ce fichier a été écrit par l'infection. En effet, en extrayant les parties `write` et `load` du graphe pour le processus `xxx-porn-movie.avi.exe`, nous pouvons voir que c'est l'infection qui écrit et charge ce fichier.

La figure 5.10 nous montre une partie des interactions faites par le processus `InstallFlashPlayer.exe` qui vient d'être exécuté par le logiciel malveillant. Nous pouvons voir que celui-ci va aussi écrire dans la corbeille de Windows.

Enfin, la figure 5.11 nous montre que c'est le processus `services.exe` qui va charger les fichiers qui ont été écrits par les deux processus malveillants. C'est à ce moment là que la charge active du logiciel malveillant est exécutée. L'exécution, par un service système, de cette charge, assure à l'infection la possibilité de pouvoir réaliser toutes les interactions qu'elle désire.

Dans cette étude, nous pouvons noter plusieurs points : tout d'abord, la dissimulation de la véritable extension par le logiciel malveillant dans le but de tromper l'utilisateur. Cette technique est assez courante pour installer des logiciels malveillants. Ensuite, l'utilisation d'un logiciel légitime pour installer la charge active de l'infection qui est aussi une méthode pour se cacher aux yeux de l'utilisateur. Ici, le logiciel malveillant peut justifier l'installation par nécessité pour lire la vidéo. Nous notons aussi l'utilisation d'un répertoire temporaire, ici la Corbeille, pour l'écriture des fichiers utilisés pour exécuter la charge active. Enfin, nous voyons l'utilisation de processus légitime, présent sur tous les systèmes Windows et tournant avec des privilèges élevés pour installer la charge active.

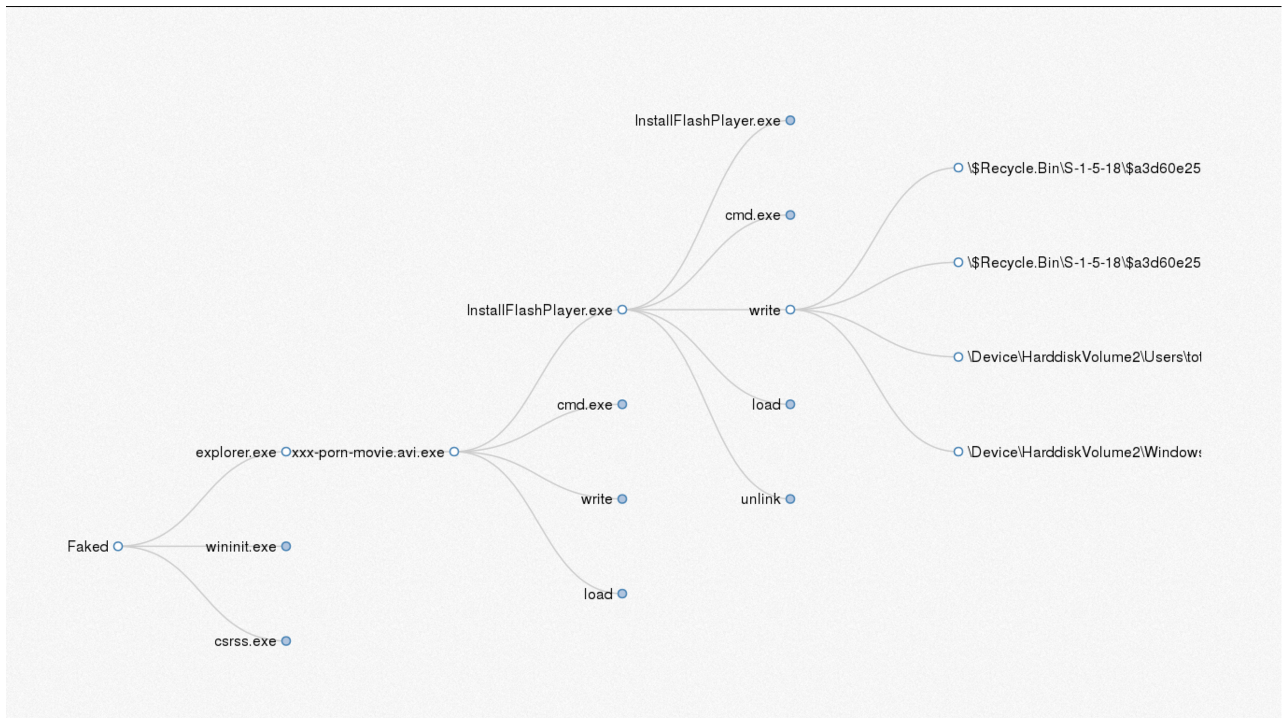

FIGURE 5.10 – `Flash player` écrit dans la Corbeille de Windows

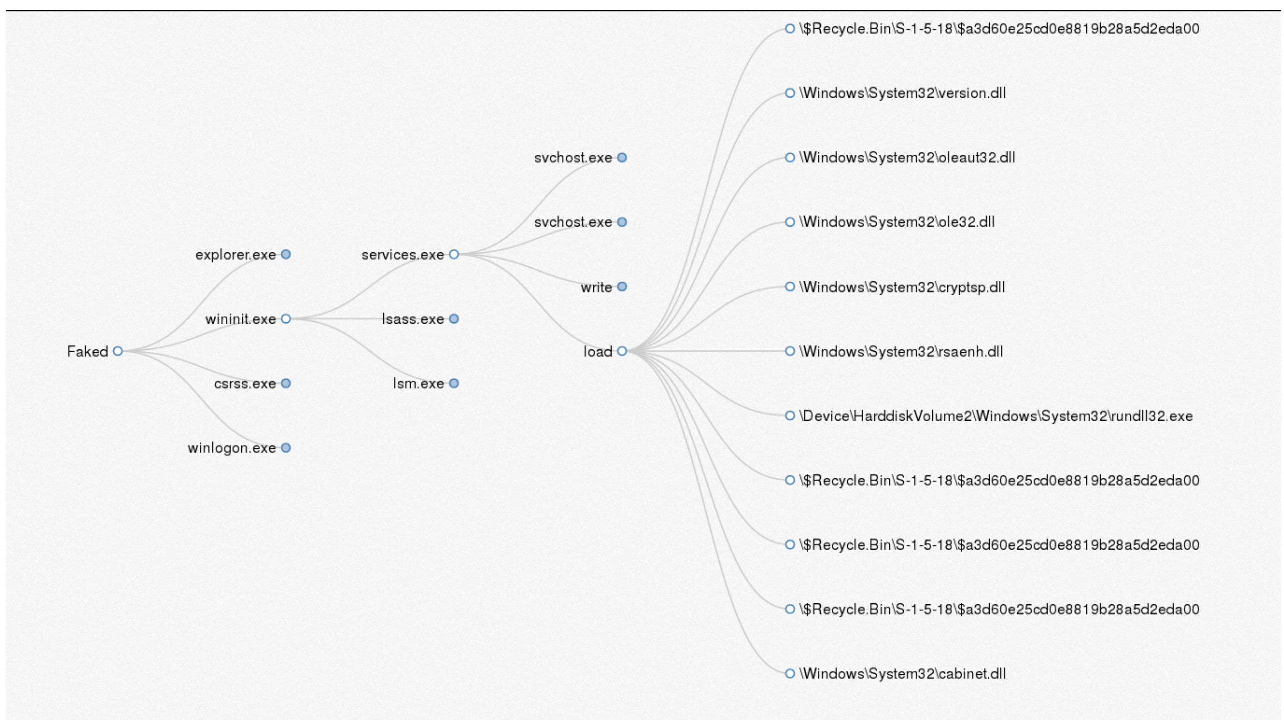

FIGURE 5.11 – `services.exe` chargeant les fichiers écrits par les logiciels malveillants.

Nous nous sommes intéressés uniquement à l'étude de l'installation de l'infection, pour connaître son cheminement et les fichiers qui ont été contaminés par l'infection.

Cette représentation permet de voir les liens entre les différentes entités du système ainsi que de connaître les interactions réalisées par chaque processus au moment de l'étude. Néanmoins, elle ne permet pas de tout représenter. En effet, dans un souci de clarté pour la représentation, nous avons fait le choix de ne pas y inclure toutes les interactions des processus ni les éléments du registre. De plus, cette représentation nous fait perdre la notion de temporalité des interactions.

Il faut noter que cette première représentation est suffisante pour comprendre le comportement global de l'infection et établir des règles de contrôle d'accès permettant de contrer explicitement son installation. Mais pour comprendre tout le cheminement de l'installation, il est nécessaire d'ajouter la notion de temps pour situer les interactions les unes par rapport aux autres.

Automates

La représentation des traces sous la forme d'automates nous permet d'exprimer la notion de temporalité des interactions que nous n'avons pas avec la représentation sous la forme de graphe. Nous allons ainsi générer un automate global, c'est-à-dire que nous allons représenter les opérations importantes réalisées par le logiciel malveillant pour en dégager un comportement. Nous affinerons ce comportement en prenant en compte des "sous automates". Grâce à ces différents éléments, nous pourrons écrire des règles de contrôle d'accès ciblant des comportements que nous considérons comme malveillants.

La figure 5.12 montre un résultat simplifié de la génération de l'automate. Ce résultat est simplifié puisque nous n'identifions qu'un nombre restreint d'actions qui conduisent à l'installation de l'infection. Nous voyons dans cet automate que le canal de communication utilisé entre les différentes entités mises en jeu dans l'installation de l'infection est une valeur dans le registre. Nous avons nommé *sample* le fichier binaire qui est à la base de l'installation.

Cette représentation simplifiée permet d'inclure les éléments qui étaient difficiles à inclure dans les graphes pour ne pas nuire à la lecture. La simplification repose sur deux éléments : tout d'abord, nous ne nommons pas explicitement les objets du système, ensuite, nous n'utilisons pas directement le temps des interactions, nous numérotons les interactions pour plus de clarté.

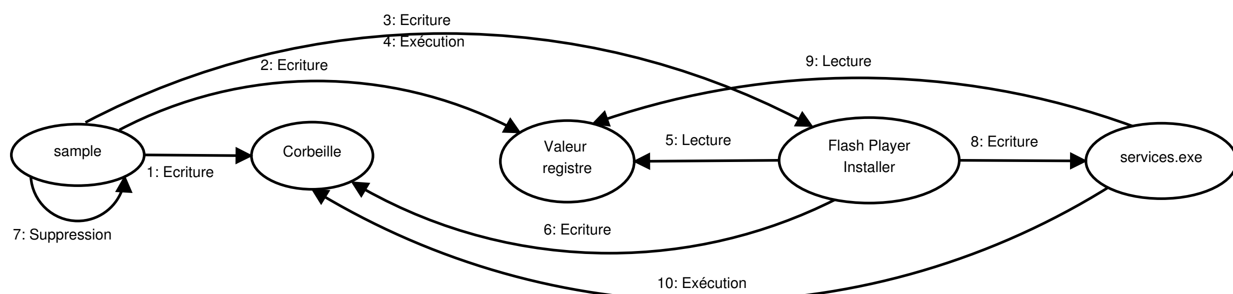

FIGURE 5.12 – Automate complet des interactions réalisées par le logiciel malveillant

L'automate met en relief la séquentialité des opérations qui se sont déroulées durant l'infection. Cette forme sert d'intermédiaire entre les graphes des processus et les diagrammes de séquence.

Même si les interactions sont numérotées, cet automate fait perdre la notion de temporalité des interactions. Nous proposons donc une représentation sous la forme d'un diagramme de séquence pour raffiner encore l'analyse comme l'illustre la figure 5.13. C'est à partir de ce diagramme que nous allons pouvoir identifier des comportements malveillants ou des séquences d'interactions propres à la détection de logiciel malveillant.

À partir de l'automate que nous avons présenté 5.12 ainsi que du diagramme de séquence 5.13, nous allons résumer le déroulement de l'installation de l'infection :

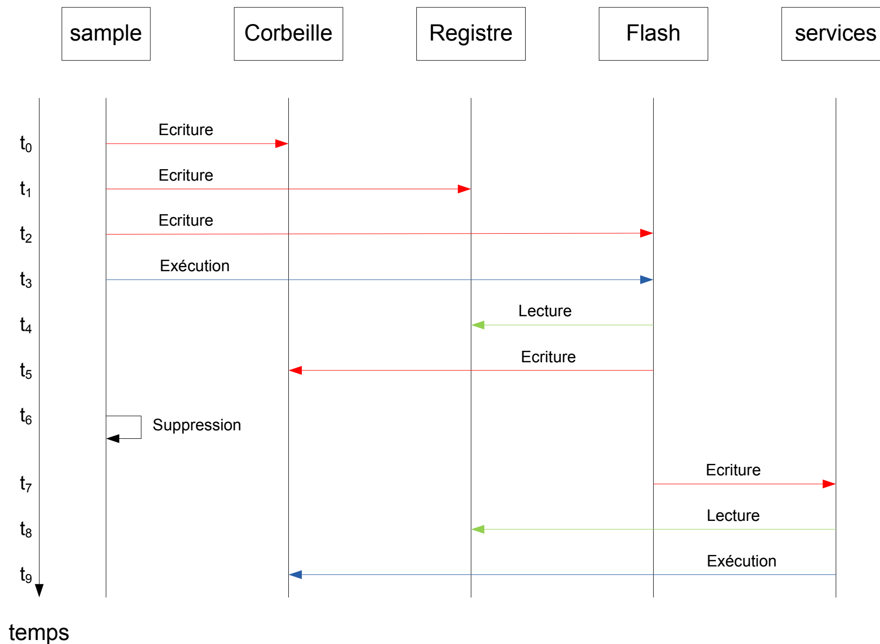

FIGURE 5.13 – Diagramme d'activité sur l'installation de l'infection

— écriture du *sample* sur le disque et exécution du fichier : ces deux phases ne sont pas représentées ici car elles peuvent varier suivant les contextes. On peut imaginer un utilisateur qui télécharge et qui l'exécute, ou encore une exploitation de vulnérabilité dans un module tiers d'un navigateur.
— écriture des éléments pour initialiser l'installation de l'infection dans un répertoire spécifique. Ce répertoire est ensuite renseigné dans le registre ;
— exécution de la charge active ;
— suppression du *sample* ;
— lecture des informations importantes pour l'installation et l'écriture des éléments permettant à l'infection de survivre ;
— exploitation d'une vulnérabilité dans un processus possédant des droits élevés.

Nous avons donc proposé trois outils permettant d'analyser les logiciels malveillants. Ces trois outils sont complémentaires en proposant chacun un degré de précision différent.

La première représentation, celle sous la forme de graphe, permet de connaître, de façon macroscopique, les interactions qui ont été faites par tous les éléments du système. On peut ainsi connaître les éléments qui ont été écrits, lus, chargés et supprimés. Nous n'avons présenté dans cette étude que les interactions faites sur le système de fichiers. Nous pouvons naturellement faire la même chose pour les interactions faites au niveau du registre.

La seconde représentation propose un raffinement de l'analyse en numérotant le début des interactions grâce à l'utilisation des `timestamps`. Cette représentation permet de connaître l'enchainement des interactions faites sur le système.

La troisième représentation ajoute un nouveau niveau de raffinement en détaillant les temps du début des interactions. Ce diagramme pourra être étendu en rajoutant les dates de fin des interactions.

Politiques de protection

Maintenant que nous connaissons le comportement du logiciel malveillant par la définition de l'automate et par la création d'un diagramme de séquence, nous pouvons écrire des politiques de protection dans le but de bloquer ces comportements malveillants.

```
1 neverallow domain recycle_bin_t:file { execute }
2 neverallow domain recycle_bin_t:process { execute }
```

Listing 5.29 – Règle bloquant l'exécution d'un élément de la corbeille

Le premier comportement que l'on peut bloquer est l'exécution d'un élément dans la corbeille de Windows. Pour ce faire, il suffit de mettre une labellisation générique sur cet élément du type `recycle_bin_t` et d'empêcher toute exécution d'un objet de ce type.

Par exemple, les règles 5.29 permettent de contrer l'exécution d'un fichier de type `recycle_bin_t`. Pour cela, nous allons utiliser une interaction spécifique de la politique par l'utilisation de l'instruction *neverallow* dans la politique de contrôle d'accès. De plus, pour que cette règle s'applique à tous les sujets du système, nous utiliserons l'instruction *domain*, qui représente tous les domaines du système.

Cette règle bloque donc explicitement l'interaction numéro 10 de notre automate 5.12.

Cependant, ces règles de contrôle d'accès peuvent se révéler difficiles à mettre en place. En effet, elles pourraient nuire à l'utilisation normale du système. Imaginons par exemple qu'un administrateur définisse la règle suivante 5.30, qui empêche toute écriture de fichier dans la corbeille, cela implique que les utilisateurs ne peuvent plus déplacer de fichiers ou de dossiers dans la corbeille. Ce qui nuit au fonctionnement d'un programme ou du système.

```
1 neverallow domain recycle_bin_t:file { write }
```

Listing 5.30 – Règle bloquant l'écriture d'un élément dans la corbeille

C'est pour cela que nous allons plutôt utiliser des propriétés de sécurité en utilisant le langage de PIGA. Nous noterons les contextes de sécurité sujets scs et les contextes de sécurité objets sco.

Voici, sous une forme simple, les interactions qui se déroulent sur le système en termes de contexte de sécurité et d'opérations élémentaires.

1. $scs_1 \rightarrow_w sco$: création d'un nouvel objet, *i.e :* le *sample* écrit dans la corbeille.
2. $scs_1 \rightarrow_w sco_1$: création d'un second objet, *i.e :* le *sample* écrit dans les fichiers temporaires l'installateur modifié de `FlashPlayer`.
3. $scs_1 \rightarrow_x sco_1 \Rightarrow_t scs_2$: exécution du second objet qui transite vers un nouveau contexte sc_2.
4. $scs_2 \rightarrow_w scs_3$: le nouveau domaine interagit avec un troisième domaine déjà présent sur le système.
5. $scs_3 \rightarrow_x sco$: ce troisième domaine va exécuter l'objet écrit par le premier domaine.

Cette énumération nous permet d'écrire directement la propriété de sécurité correspondante 5.31.

```
1 define zeroaccess_behavior ( $sc1 IN SCS ) [
2     Foreach $eo1 IN is_write_like(IS), Foreach $eo2 IN is_write_like(IS),
          Foreach $eo3 IN is_execute_like(IS),
3           Foreach $oe4 IN is_write_write(IS), Foreach $oe5 IN is_execute_like(
                IS),
4     Foreach $sc2 IN $SCS, Foreach $sc3 IN $SCS,
5     Foreach $sco1 IN $SCO, Foreach $sco IN $SCO,
6     Foreach $a1 IN ACT, Foreach $a2 IN ACT
7         TQ { ( [ $a2 := $sc3 -> { $eo5 } $sco ] o $sc2 -> { $eo4 } $sc3 )
8             o ( $sc1 -> { $eo3 } $sco1 o $sc1 -> { $eo2 } $so1 o [ $a1 := $sc1
                  -> { $eo1 } $sco ] ) } ,
9                 { INHERIT($a2 , $a1) };
```

```
10
11 ];
```

Listing 5.31 – Propriété de sécurité pour PIGA contrant l'installation d'une variante de *ZeroAccess*

Grâce à cette propriété, nous sommes capables de contrôler de manière un scénario complet.

Nous avons donc proposé deux méthodes pour contrer l'installation de ce logiciel malveillant. La première méthode consiste à bloquer une interaction directe qui ne semble pas légitime faite par le logiciel malveillant. Même si cette solution est efficace, sa portée sur le fonctionnement du système pourrait se révéler très préjudiciable. En effet, les règles d'accès créées pourraient bloquer le système. C'est pour cela que nous avons proposé une solution basée sur l'utilisation d'un second moniteur de référence, qui offre la possibilité de bloquer un scénario complet plutôt qu'une interaction précise.

5.4 Discussion

Dans ce chapitre, nous avons détaillé les choix que nous avions fait pour répondre aux problèmes soulevés dans la partie formalisation.

En ce qui concerne la problématique de désignation des contextes sous Windows, nous avons fait le choix, pour la construction des noms symboliques absolus indépendants de la localisation d'utiliser les variables d'environnement. Elles ont l'avantage d'être à la fois communes à tous les systèmes Windows et d'être gérées par le système. De plus, elles peuvent être étendues par l'administrateur pour répondre à des problèmes spécifiques.

Cette solution nous a permis de répondre à un second problème plus lié à l'implantation. En effet, nous avons dû résoudre la problématique de stockage des contextes de sécurité pour les objets dans le modèle de protection basé sur DTE. Comme nous ne pouvions utiliser les attributs étendus du système de fichiers comme le fait SELinux, nous avons mis en place un système de labellisation dynamique. Cette méthode présente l'avantage d'être plus simple à administrer et plus portable car ne nécessitant pas de labelliser les systèmes des fichiers.

Nous avons ensuite proposé deux implantations des méthodes de détournement que nous avons décrites dans la partie formalisation. Ces deux méthodes nous ont permis de tester et de valider nos politiques de protection pour les deux modèles PBAC et DTE. Nous avons ensuite fait le choix de poursuivre nos expérimentations en utilisant le modèle de protection DTE.

Nos expérimentations ont porté sur deux aspects. Le premier point montre que nous sommes capables d'associer notre moniteur Windows au moniteur PIGA pour contrôler efficacement les scénarios complets. Le second concerne la protection contre les logiciels malveillants. Une architecture dédiée permet l'envoi des traces de notre moniteur à un serveur limitant ainsi la compromission de l'analyse. L'analyse permet de comprendre le scénario complet et de déduire une politique pour notre moniteur ou pour PIGA qui empêche le scénario d'attaque. La solution PIGA propose une méthode optimiste qui limite moins les activités légitimes.

5.4.1 Pertinence/complétude de la solution

Notre solution, basée sur une architecture modulaire est capable d'implanter deux modèles de protection. Le premier est basé sur le modèle PBAC et le second sur le modèle DTE. Ces deux modèles permettent de définir des politiques de contrôle d'accès fines et précises. Chaque ressource du système est parfaitement identifiée.

Notre solution facilite la maintenabilité du mécanisme de contrôle d'accès. Le mécanisme de création de politique basé à la fois sur un mode d'apprentissage auquel s'ajoute un outil de création de politique de contrôle d'accès, facilite le travail de l'administrateur. Quel que soit le modèle de

protection utilisé, il est possible d'avoir une couche d'abstraction efficace des ressources et ainsi de pouvoir porter facilement des politiques d'un système à un autre.

Que ce soit en modifiant la table contenant tous les appels système ou par l'ajout d'un *filter-driver*, nous sommes capables de gérer finement les accès, que ce soit sur le registre, au niveau réseau et ou sur le système de fichiers. Non seulement nous pouvons vérifier les accès en prétraitement, mais aussi en post-traitement puisque nous détournons entièrement le flux d'exécution, chose que ne font pas encore les MAC tels que SELinux.

Grâce à cette implantation, nous contrôlons toutes les entrées/sorties des processus. Nous contrôlons aussi les méthodes de synchronisations entre `threads`. Ces évènements passent par des appels système que nous contrôlons.

5.4.2 Performances

Même si l'objectif de nos expérimentations n'était pas les performances, nous avons évalué la latence que génère notre implantation. Nous avons fait ce test de performance sur la version qui modifie la table des appels système.

Nous avons ainsi pu noter qu'il y avait une latence générée au démarrage des applications par notre mécanisme de contrôle d'accès. Cela s'explique par les nombreuses interactions faites sur le système de fichiers réalisées par l'application pour charger ses bibliothèques, créer ses fichiers spécifiques, accéder au registre pour récupérer ses informations de configuration, etc.

Cette latence, que nous avons notée et estimée à un ajout de 2 secondes en moyenne par rapport au temps de lancement normal de l'application, peut facilement être réduite en introduisant les mêmes optimisations que pour SELinux. Par exemple, l'ajout d'un système de cache dans notre driver SEWINDOWS pourrait accélérer la prise de décision. Ainsi, il ne serait pas nécessaire d'aller consulter la politique SEWINDOWS à chaque interaction faite par le même contexte sujet.

Une seconde optimisation serait de ne pas vérifier à chaque fois les accès pour certains types d'interactions, comme les opérations de lecture ou d'écriture. Par exemple, lorsqu'un sujet demande l'accès en lecture sur un objet, l'accès est vérifié une première fois dans la politique. Puis, tant que les contextes de sécurité mis en jeu ne sont pas modifiés, nous pouvons considérer qu'il est inutile de vérifier de nouveau l'accès. Il faudrait vérifier que cette optimisation ne nuit pas à la sécurité du système et qu'il n'existe pas de solution pour la contourner. Il faut noter que cette optimisation a été mise en place dans SELinux.

Enfin, une troisième optimisation concerne le parcours de la politique par le serveur de sécurité. Nous avons développé une première version où le moniteur mémorise la politique sous la forme de chaîne de caractères. Nous avons ensuite mis en place un système basé sur des tables de hashages. Une méthode pour optimiser la recherche dans la politique serait d'avoir une table par sujet, ainsi, il ne serait plus nécessaire de parcourir la politique entièrement à chaque fois.

Chapitre 6

Conclusion

Les enjeux de cette thèse étaient doubles. D'une part, il s'agissait d'améliorer la sécurité des systèmes de calcul intensif basés sur Linux. D'autre part, des solutions devaient être apportées pour que les postes de travail Windows puissent garantir des objectifs de sécurité. Le fait de rassembler ces deux points présentait déjà une difficulté technique puisqu'il fallait connaitre deux systèmes d'exploitation et maîtriser le développement dans leurs noyaux respectifs. Cet aspect technique constituait en soi un pari. Un des objectifs était donc d'offrir un cadre commun pour protéger des contextes d'usages aussi différents que le calcul intensif et les postes de travail Windows. Définir une approche extensible était une première difficulté. De plus, il était nécessaire de s'intéresser en parallèle à la sécurité et aux performances, ce qui est en soi assez antinomique. Enfin, il fallait s'intéresser aux vulnérabilités des postes de travail et offrir des méthodes avancées d'analyse et de protection. Il aurait été possible de faire une thèse sur chacun de ces points pris séparément. Il n'en reste pas moins que s'intéresser de façon globale à ces domaines en dégageant correctement les problèmes et en proposant des solutions extensibles constitue déjà un objectif ambitieux et intéressant. Au final, nous répondons globalement à ces différents points d'étude en définissant un cadre commun à la protection du calcul intensif et des postes de travail.

En introduction, nous avons ainsi décrit le problème de sécurité, à savoir contrôler les interactions directes et être capable de contrôler les scénarios d'attaque complets. Nous avons aussi synthétisé les manques dans ces domaines, à savoir l'absence de modèle général, non seulement pour observer les appels système, mais aussi pour répartir efficacement différents observateurs afin de détecter et prévenir des scénarios d'attaque complets. Nous nous sommes intéressés aux observateurs appelés moniteurs de référence et capables de garantir l'intégrité et la confidentialité. Nous avons considéré aussi le besoin d'une méthode pour répartir ces moniteurs, ainsi que mesurer et comparer les performances de différentes solutions. Enfin, nous avons considéré, d'une part, la nécessité d'un observateur constituant un moniteur de référence portable pour contrôler les interactions directes, et d'autre part, d'une implantation de ce moniteur pour Windows afin d'analyser et prévenir les scénarios d'activité des logiciels malveillants.

Dans le chapitre 2, nous avons établi l'état de l'art. Il montre à la fois le manque de modèle générique d'observateurs, les faiblesses des moniteurs de référence existants et le manque de modèle et de solutions pour répartir différents observateurs et évaluer leurs performances de façon précise et extensible. Cet état de l'art étaye les points adressés par notre étude et justifie la réalité des difficultés que nous avons listées en introduction. Il montre qu'il y a un réel manque de moniteurs répartis pour le calcul intensif et de solutions pour les postes de travail Windows.

Le chapitre 3 a proposé une définition générique de la notion d'observateur capable de capturer les appels système des différents processus pour contrôler les accès aux ressources du système d'exploitation. Nous avons distingué trois modes de sécurité : requête/réponse, évidence et notification. Ils couvrent la majorité des approches permettant de détecter ou protéger les activités.

Cette notion d'observateur est très générale et correspond à un large ensemble de mécanismes de sécurité (pare-feu, contrôle d'accès des processus, antivirus, anti*malware*, détection d'intrusion...). Nous avons définis un type particulier d'observateur, le moniteur de référence, qui nous intéresse en priorité. Ensuite, nous avons proposé un modèle conceptuel qui caractérise la répartition des observateurs en terme d'association, de localisation et de redondance. Les deux types d'association, cascade et notification, permettent de décrire le couplage de différents observateurs. En pratique, le type cascade est le plus représentatif car il couvre bien la façon habituelle de coopérer des observateurs. Les quatre localisations distinguent les observateurs colocalisés, distants, parallèles et partagés. Ces différentes localisations visent à classifier les méthodes de répartition des observateurs et de leurs clients. La redondance concerne les méthodes de tolérance aux pannes d'un observateur. Pour la partie poste de travail, nous avons proposé une politique pour contrôler les accès directs de façon obligatoire (MAC). Cette politique est générique, mais correspond bien aux modèles DTE et PBAC adoptés sous Unix. Nous avons défini une façon de mettre en œuvre cette politique pour Windows en offrant un mécanisme de désignation des processus et des ressources qui simplifie l'administration des politiques. Nous avons proposé différentes méthodes pour détourner les appels système des processus sous Windows afin de pouvoir contrôler leurs activités. Ainsi, par la conjonction du modèle de politique et d'un détournement, nous sommes capables d'offrir un modèle conceptuel de moniteur de référence qui est bien adapté à Windows.

Le chapitre 4 a étudié la répartition des observateurs en environnement de calcul intensif. Nous avons proposé d'abord un objectif de mesure précise des performances. Pour cela, nous nous sommes concentrés sur la répartition de deux observateurs en mode cascade en considérant qu'ils peuvent être colocalisés ou distants. Nous avons proposé un calcul de la combinatoire des évaluations et établissons la liste de mesures à effectuer. Nous avons proposé ainsi différentes mesures globales et mesures détaillées. Nous pouvons déduire le temps de communication entre les deux observateurs et comparer les performances pour les combinaisons des différentes approches. Une mise en œuvre du mode distant à haute performance est proposée en utilisant des technologies InfiniBand. L'efficacité repose sur deux coupleurs InfiniBand pour les communications entre les deux observateurs. Nous avons proposé un protocole de mesure des performances reposant sur deux logiciels de test. Le premier logiciel est le benchmark `Linpack` pour le HPC. Ce benchmark n'est pas complètement représentatif des codes de calcul du CEA mais constitue le meilleur des cas puisqu'il fait peu d'opérations d'entrée/sortie. Le second est un logiciel développé spécifiquement pour stresser le système en réalisant un grand nombre d'entrées/sorties avec des politiques SELinux et PIGA exigeantes en temps de calcul. On peut considérer qu'il s'agit donc du pire cas. La réalité se situant entre ces deux cas, nous avons montré qu'il est possible de limiter le surcoût à 10% pour les nœuds de calcul pour le mode détection et 25% pour le mode protection. Des optimisations supplémentaires permettront de pouvoir passer en dessous de 5% et 10% respectivement pour le mode détection et protection. Si ces résultats restent discutables, ils montrent la possibilité d'un faible surcoût au regard du contrôle de scénarios d'attaque complets. L'approche est efficace en termes de sécurité puisqu'elle réalise un contrôle optimiste qui ne limite pas les activités légitimes.

Dans le chapitre 5, nous avons décrit la mise en œuvre d'un moniteur de référence pour Windows autorisant l'analyse et le contrôle des logiciels malveillants. Nous avons proposé une implantation de la désignation des ressources basée sur les variables d'environnement. Cette méthode calcule dynamiquement les contextes de sécurité et évite donc d'avoir à les stocker au niveau des systèmes de fichiers, simplifiant ainsi l'administration. Nous utilisons le moniteur en mode détection pour transférer de façon sûre les accès observés à un serveur. Cette approche a pour but d'analyser le scénario d'attaque complet via différentes représentations graphiques. Ainsi, nous sommes capables de décrire le déroulement temporel et logique du scénario. Nous montrons que nous pouvons définir une politique pour notre moniteur, ou pour le moniteur PIGA, qui empêche

l'attaque observée. Le contrôle des interactions directes par notre moniteur est plus immédiat, mais présente le risque de limiter les activités légitimes des processus. A contrario, l'approche PIGA est plus optimiste puisqu'elle peut bloquer le scénario à un point précis de son avancement. Nous avons développé deux méthodes de détournement des appels système, la seconde étant davantage portable. Globalement, nous livrons ainsi un moniteur portable aussi bien en termes de détournement que de politique pour des environnements Windows 7 hétérogènes.

6.1 Perspectives

Amélioration des performances de PIGA Nous avons montré au cours du chapitre 4 que l'intégration de SELinux dans les environnements HPC était aujourd'hui possible, notamment grâce aux différentes optimisations réalisées. Cette intégration est possible car le surcoût dû à SELinux est de l'ordre d'une μs par appel système. Même si le déport par Infiniband du moniteur PIGA améliore les performances, des optimisations supplémentaires sont nécessaires pour passer en dessous d'un surcoût de 5% pour les nœuds de calcul. C'est pour cela qu'il est nécessaire de travailler à l'optimisation de PIGA.

Cela peut passer par la réécriture complète de la partie PIGA-UM pour qu'elle soit résidente en espace noyau. Actuellement, la partie PIGA-UM est écrite en Java, qui est une technologie consommatrice de ressources. En modifiant le code avec un langage moins consommateur, il serait alors possible de co-localisé PIGA-UM sur un nœud client. De plus, comme nous l'avons présenté au cours de notre étude, nous avons utilisé PIGA avec des politiques de sécurité extrêmement génériques capable de couvrir un large ensemble de scénarios complets. Cette large couverture n'est pas spécifique aux environnements HPC, il est donc nécessaire de travailler ces propriétés pour qu'elles soient spécifiques à ces environnements. Grâce aux nouvelles propriétés, PIGA aurait un plus petit nombre d'activités à surveiller et ces performances en seront améliorées.

Enfin, les compressions des signatures proposées dans [Clairet *et al.*, 2012] devraient diminuer de façon importante le temps de traitement et l'occupation mémoire.

Observateurs dans les systèmes répartis Nous avons commencé à mettre en place le mode de redondance maître-esclave sur l'architecture que nous avons proposée dans le chapitre 4. Il est nécessaire de poursuivre les travaux que nous avons commencés dans le but d'avoir un système tolérant aux différentes pannes.

Nous avons concentré notre étude sur le mode distant pour le second observateur. Nous pouvons maintenant orienter nos travaux sur les modes parallèles et partagés des observateurs, dans un premier temps pour résoudre le problème de dimensionnement du mécanisme de protection, et dans un second temps pour pouvoir corréler les requêtes reçues par le second observateur. En effet, actuellement, le mode distant impose d'avoir une machine dédiée par observateur distant (dans notre étude PIGA), ce qui pose un problème de dimensionnement de l'architecture de protection puisque si nous voulons protéger 1 000 nœuds, il est nécessaire d'avoir 1 000 serveurs de sécurité pour une localisation en mode distant. En travaillant sur les modes parallèles et partagés, nous pourrions ainsi réduire le nombre de serveurs de sécurité utilisés dans notre architecture de protection. De plus, il serait alors possible d'écrire des propriétés de sécurité adaptées aux systèmes répartis.

Expérimentations à plus grande échelle de notre moniteur SEWINDOWS La première extension pour les systèmes Windows serait la mise en place d'expérimentations à grande échelle pour notre mécanisme de protection. Les objectifs seraient de corriger les limites de nos implantations, de définir des politiques de plus en plus complètes ainsi que de connaître précisément l'impact sur les performances du système.

De plus, il serait intéressant de travailler sur les contextes de sécurité. Dans la version actuelle, seuls les types et les domaines sont traités par notre moniteur, nous pourrions développer le contrôle d'accès en nous basant sur les SID et sur les rôles.

Association des observateurs sur les systèmes Windows A partir des modèles que nous avons définis dans le chapitre 3 de ce mémoire, il faudrait, par la suite, associer notre observateur avec un autre observateur réalisant des calculs plus complexes. Au cours de notre étude, nous avons réalisé cette composition de façon *offline* en rejouant les traces générées dans PIGA. Maintenant, il faudrait les faire fonctionner tous les deux sur un même système.

De plus, les travaux que nous avons présentés sous Linux, visant à utiliser des technologies du HPC pour déporter de manière efficace le second observateur, pourraient aussi être utilisés sur les systèmes Windows. En effet, le **proxy** que nous avons développé est en mesure de réaliser les mêmes actions que sur les systèmes Linux. Par extension, l'intégration de ce modèle de protection pourrait être fait dans les architectures de *cloud* basées sur les technologies de Microsoft.

Extension de la répartition à d'autres systèmes d'exploitation Nous avons présenté la notion d'observateur pour les systèmes Linux et Windows puis nous avons défini la notion de répartition. Nous avons montré au cours de notre étude que nous pouvions appliquer ces deux notions sur ces deux systèmes et dans des environnements hétérogènes. Nous pourrions maintenant étendre ces notions à d'autres systèmes d'exploitation tels que les les systèmes Android et les OSX.

De plus, comme les systèmes Android ont pour base les systèmes Linux, nous pourrions déployer les deux observateurs que nous avons testés, à savoir SELinux et PIGA.

Chapitre 7

Bibliographie

[Alexander Kjeldaas, 1998] ALEXANDER KJELDAAS (1998). Linux Capability FAQ v0.1. `http://www.uwsg.indiana.edu/hypermail/linux/kernel/9808.1/0178.html`.

[Anderson, 1980] ANDERSON, J. (1980). Computer security threat monitoring and surveillance. Rapport technique, James P. Anderson Company, Fort Washington, Pennsylvania.

[Anderson, 1972] ANDERSON, J. P. (1972). Computer Security technology planning study. Rapport technique, Deputy for Command and Management System, USA.

[ANR, 2009] ANR (2009). Anr sec&si. `http://www.agence-nationale-recherche.fr/programmes-de-recherche/appel-detail/programme-systemes-embarques-et-grandes-infrastructures-defi-securite-systeme-d-exploitation-cloisonne-et-securise-pour-l-internaute-2008/`.

[Bell et La Padula, 1973] BELL, D. E. et LA PADULA, L. J. (1973). Secure computer systems : Mathematical foundations and model. Technical Report M74-244, The MITRE Corporation, Bedford, MA.

[Biba, 1975] BIBA, K. J. (1975). Integrity considerations for secure computer systems. Technical Report MTR-3153, The MITRE Corporation.

[Bishop, 2003] BISHOP, M. (2003). *Computer Security Art and Science*. Numéro ISBN 0201440997. Addison-Wesley Professional.

[Blanc, 2006] BLANC, M. (2006). *Sécurité des systèmes d'exploitation répartis : architecture décentralisée de méta-politique pour l'administration du contrôle d'accès obligatoire.* These, Université d'Orléans.

[Blanc *et al.*, 2014] BLANC, M., BOUSQUET, A., BRIFFAUT, J., CLEVY, L., GROS, D., LEFRAY, A., ROUZAUD-CORNABAS, J., TOINARD, C. et VENELLE, B. (2014). Mandatory access protection within cloud systems. *In* NEPAL, S. et PATHAN, M., éditeurs : *Security, Privacy and Trust in Cloud Systems*, pages 145–173. Springer Berlin Heidelberg.

[Blanc *et al.*, 2011] BLANC, M., BRIFFAUT, J., TOINARD, C. et GROS, D. (2011). PIGA-HIPS : Protection of a shared HPC cluster. *International journal on advances in security*, 4(1):44–53.

[Blanc *et al.*, 2012] BLANC, M., GROS, D., BRIFFAUT, J. et TOINARD, C. (2012). PIGA-Windows : contrôle des flux d'information avancés sur les systèmes d'exploitation Windows 7. *In MajecSTIC 2012*, Villeneuve d'Ascq, France.

[Blanc *et al.*, 2013a] BLANC, M., GROS, D., BRIFFAUT, J. et TOINARD, C. (2013a). Mandatory access control with a multi-level reference monitor : PIGA-cluster. *In ACM CLHS '13 Procee-*

dings of the first workshop on Changing landscapes in HPC security, pages 1–8, New-York, United States. ACM.

[Blanc *et al.*, 2013b] BLANC, M., GROS, D., BRIFFAUT, J. et TOINARD, C. (2013b). PIGA-Cluster : a distributed architecture integrating a shared and resilient reference monitor to enforce mandatory access control in the HPC environment. *In SHPCS - 8th International Workshop on Security and High Performance Computing Systems - 2013*, Helsinki, Finland.

[Blanc et Lalande, 2012] BLANC, M. et LALANDE, J.-F. (2012). Improving Mandatory Access Control for HPC clusters. *Future Generation Computer Systems*, pages –.

[Boebert et Kain, 1985] BOEBERT, W. E. et KAIN, R. Y. (1985). A practical alternative to hierarchical integrity policies. *In The 8th National Computer Security Conference*, pages 18–27, Gaithersburg, MD, USA.

[Briffaut, 2007] BRIFFAUT, J. (2007). *Formalization and guaranty of system security properties : application to the detection of intrusions*. Thèse de doctorat, Université d'Orléans. SDS.

[Briffaut *et al.*, 2009] BRIFFAUT, J., LALANDE, J.-F. et TOINARD, C. (2009). Formalization of security properties : enforcement for MAC operating systems and verification of dynamic MAC policies. *International journal on advances in security*, 2(4):325–343. ISSN : 1942-2636.

[Casey Schaufler , 2008] CASEY SCHAUFLER (2008). The Simplified Mandatory Access Control Kernel . *White Paper*, pages 1–11.

[Clairet *et al.*, 2012] CLAIRET, P., BERTHOMÉ, P. et BRIFFAUT, J. (2012). Compression de signatures pour PIGA IDS. *In* ETIEN, A., éditeur : *9ème édition de la conférence MAnifestation des JEunes Chercheurs en Sciences et Technologies de l'Information et de la Communication - MajecSTIC 2012 (2012)*, Villeneuve d'Ascq, France. Nicolas Gouvy.

[Clark et Wilson, 1987] CLARK, D. D. et WILSON, D. R. (1987). A Comparison of Commercial and Military Computer Security Policies. *Proc. IEEE Symp. Computer Security and Privacy, IEEE CS Press*, pages 184–194.

[Corporation, 2003] CORPORATION, N. D. (2003). Tomoyo. `http://tomoyo.sourceforge.jp/index.html.en`.

[Darivemula *et al.*, 2006] DARIVEMULA, A., BOX, C., TIKOTEKAR, A. et POURZANDI, M. (2006). Work in progress : Rass framework for a cluster-aware selinux. *Cluster Computing and the Grid, IEEE International Symposium on*, 2:29.

[Ferraiolo et Kuhn, 1992] FERRAIOLO, D. F. et KUHN, D. R. (1992). Role-based access controls. *In 15th National Computer Security Conference*, pages 554–563, Baltimore, MD, USA.

[Focardi et Gorrieri, 2001] FOCARDI, R. et GORRIERI, R. (2001). Classification of security properties (part i : Information flow).

[Gros *et al.*, 2012] GROS, D., TOINARD, C. et BRIFFAUT, J. (2012). Contrôle d'accès mandataire pour Windows 7. *In SSTIC 2012*, pages 266–291, Rennes, France.

[Hagimont et J.Mossière, 1996] HAGIMONT, D. et J.MOSSIÈRE (1996). Problèmes de désignation, de localisation et d'accès dans les systèmes répartis à objets . Rapport technique.

[Harrison *et al.*, 1976] HARRISON, M. A., RUZZO, W. L. et ULLMAN, J. D. (1976). Protection in operating systems. *Communications of the ACM*, 19(8):461–471.

[Hoglund et Butler, 2005] HOGLUND, G. et BUTLER, J. (2005). *Rootkits : Subverting the Windows Kernel*. Addison-Wesley Professional.

[Hunt et Brubacher, 1999] HUNT, G. et BRUBACHER, D. (1999). Detours : Binary interception of win32 functions. *In Proceedings of the 3rd Conference on USENIX Windows NT Symposium - Volume 3*, WINSYM'99, pages 14–14, Berkeley, CA, USA. USENIX Association.

[ITSEC, 1991] ITSEC (1991). Information Technology Security Evaluation Criteria (ITSEC) v1.2. Technical report.

[Labs, 2005] LABS, C. (2005). Core force user's guide. pages 1–2.

[Lampson, 1969] LAMPSON, B. W. (1969). Dynamic protection structures. *In AFIPS Fall Joint Computer Conference (FJCC 1969)*, volume 35, pages 27–38, Las Vegas, Nevada, USA. AFIPS Press.

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

[Lampson, 1973] LAMPSON, B. W. (1973). A note on the confinement problem. *Commun. ACM*, 16(10):613–615.

[Leangsuksun et Haddad, 2004] LEANGSUKSUN, C. et HADDAD, I. (2004). Building highly available hpc clusters with ha-oscar. *In Cluster Computing, 2004 IEEE International Conference on*, page 2.

[Leangsuksun *et al.*, 2005] LEANGSUKSUN, C., TIKOTEKAR, A., POURZANDI, M. et HADDAD, I. (2005). Feasibility study and early experimental results towards cluster survivability. *In Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid - Volume 01*, CCGRID '05, pages 77–81, Washington, DC, USA. IEEE Computer Society.

[M. Fox et Thomas, 2003] M. FOX, J. Giordano, L. S. et THOMAS, A. (2003). Selinux and grsecurity : a side-by-side comparison of mandatory access control and access control list implementations. Rapport technique.

[McAfee, 2013] MCAFEE (2013). ZeroAccess Rootkity. Rapport technique, McAfee.

[Microsoft, 2009] MICROSOFT (2009). Applocker. `http://technet.microsoft.com/en-us/library/dd759117.aspx`.

[Microsoft, 2013] MICROSOFT (2013). Windows integrity mechanism design. *In The Microsoft Developper Network*.

[Naldurg *et al.*, 2006] NALDURG, P., SCHWOON, S., RAJAMANI, S. et LAMBERT, J. (2006). 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. ACM.

[OASIS, 2013] OASIS (2013). Xacml. `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml`.

[Pourzandi *et al.*, 2002] POURZANDI, M., HADDAD, I., LEVERT, C., ZAKRZEWSKI, M. et DAGENAIS, M. (2002). A distributed security infrastructure for carrier class linux clusters.

[Richard Ward, 2006] RICHARD WARD, Jeffrey Hamblin, P. B. (2006). Mandatory integrity control.

[Rouzaud-Cornabas, 2010] ROUZAUD-CORNABAS, J. (2010). *Formalisation de propriétés de sécurité pour la protection des systèmes d'exploitation*. Thèse de doctorat.

[Saltzer et Schroeder, 1975] SALTZER, J. et SCHROEDER, M. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278 – 1308.

[Sandhu, 1988] SANDHU, R. S. (1988). The schematic protection model : Its definition and analysis for acyclic attenuating schemes. *Journal of the ACM*, 35(2):404–432.

[Sandhu, 1992] SANDHU, R. S. (1992). The Typed Access Matrix Model. *In Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 122–136, Oakland, CA, USA. IEEE.

[Sandhu *et al.*, 1996] SANDHU, R. S., COYNE, E. J., FEINSTEIN, H. L. et YOUMAN, C. E. (1996). Role-based access control models. *Computer*, 29:38–47.

[Soshi *et al.*, 2004] SOSHI, M., MAEKAWA, M. et OKAMOTO, E. (2004). The dynamic-typed access matrix model and decidability of the safety problem. *IEICE Transactions*, 87-A(1):190–203.

[Spencer *et al.*, 1998] SPENCER, R., SMALLEY, S., LOSCOCCO, P., (national SECURITY AGENCY), P. L., HIBLER, M., LEPREAU, J. et ANDERSEN, D. (1998). The flask security architecture : System support for diverse security policies. *In in Proceedings of The Eighth USENIX Security Symposium*, pages 123–139.

[Spender, 2003] SPENDER (2003). Grsecurity feature. `https://grsecurity.net/features.php`.

[Spengler, 2002] SPENGLER, B. (2002). Detection, prevention, and containment : A study of grsecurity. *In In Libre Software Meeting 2002 (LSM2002)*, Bordeaux, France.

[Takeda, 2009] TAKEDA, K. (2009). Tomoyo linux overview. *security mini-conf*.

[TCSEC, 1985] TCSEC (1985). Trusted Computer System Evaluation Criteria. Technical Report DoD 5200.28-STD, Department of Defense.

[Team, 2012] TEAM, P. (2012). 20 Years of PaX. pages 1–20, Rennes, France. SSTIC 2012.

[Wang *et al.*, 2008] WANG, X., LI, Z., LI, N. et CHOI, J. Y. (2008). Precip : Towards practical and retrofittable confidential information protection. *In In 16th Annual Network and Distributed System Security Symposium*.

[Wright *et al.*, 2002] WRIGHT, C., COWAN, C., SMALLEY, S., MORRIS, J. et KROAH-HARTMAN, G. (2002). Linux security modules : General security support for the linux kernel. *In Proceedings of the 11th USENIX Security Symposium*, pages 17–31, Berkeley, CA, USA. USENIX Association.

Première partie

Annexes

Annexe A

Configuration pour le logiciel `linpack` et utilisation

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (6=stdout,7=stderr,file)
1            # of problems sizes (N)
26752    Ns
1            # of NBs
128          NBs
0            PMAP process mapping (0=Row-,1=Column-major)
1            # of process grids (P x Q)
2            Ps
4            Qs
16.0         threshold
3            # of panel fact
0 1 2        PFACTs (0=left, 1=Crout, 2=Right)
2            # of recursive stopping criterium
2 4          NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
3            # of recursive panel fact.
0 1 2        RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
0            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
0            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
64           swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
```

Listing A.1 – Fichier de configuration pour le logiciel `linpack`

Annexe B

Code utilisé pour la réalisation des tests de performances

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <string.h>

int main(){
        struct timespec t1,t2;
        FILE * file =NULL;
        char buffer[42];
        char fichier[102];

        while(1)
        {
                bzero(buffer, sizeof(buffer));
                bzero(fichier, sizeof(fichier));
                srand(time(NULL));

                clock_gettime(CLOCK_MONOTONIC, &t1);

                file = fopen("/dev/urandom", "rb+");
                if(file ==NULL) exit(-1);

                fread(buffer, sizeof(buffer),1,file);
                fclose(file);

                strcat(fichier,"/local/test/");
                sprintf(fichier,"%s%i", fichier,rand());
                sprintf(fichier,"%s%li", fichier,t1.tv_nsec);
                sprintf(fichier,"%s%i", fichier,rand());

                file =NULL;
                file =fopen(fichier, "wb+");
                if(file == NULL) exit(-6);

                fwrite(buffer, sizeof(buffer), 1, file);
                fclose(file);

                clock_gettime(CLOCK_MONOTONIC, &t2);
                printf("time : %li.%li\n", t1.tv_sec, t1.tv_nsec);
                printf("time : %li.%li\n", t2.tv_sec, t2.tv_nsec);
        }
        return 1;
}
```

Listing B.1 – Code utilisé pour les tests de performances

174

Annexe C

Les techniques de détournements

Il existe plusieurs techniques pour détourner les appels système sur les systèmes Windows. Ces techniques sont dangereuses car elle peuvent endommager le système en générant des BSOD (*Blue Screen of Death*), c'est-à-dire que le noyau crashe, et ainsi rendre le système complètement instable ce qui rend ces techniques illégitimes. Il existe cependant des techniques de détournement légitimes, c'est-à-dire qu'elles sont nativement intégrées au système. Nous détaillerons dans cette partie trois techniques spécifiques. Nous avons choisi ces techniques car ce sont les plus connues et les plus utilisées à ce jour. Ces trois techniques sont des techniques destinées à être utilisées en espace noyau. Il existe d'autres techniques en espace utilisateur, mais leur efficacité ainsi que leur portée sur le système n'est pas la même.

C.1 Détournement de la table des appels système

La première technique est le détournement de la table des appels système. Cette méthode est apparue sur les systèmes Windows avec les premiers *rootkits* nommés `rustock` [1].

Avantages et portée Cette première technique est plutôt simple à mettre en place avec une grande portée puisque toutes les applications utilisent la table des appels système pour réaliser une action sur le système. Elle gère les accès au système de fichiers, les accès au registre, les actions entre les processus comme la création de *pipe nommé* mais aussi les accès réseau.

L'un des principaux avantages de cette technique est qu'elle est maintenant assez documentée. On peut par exemple retrouver les explications techniques pour la mettre en œuvre dans le livre [Hoglund et Butler, 2005]. De plus, lorsqu'elle est correctement mise en place, elle peut permettre l'empilement des logiciels de sécurité, c'est-à-dire que plusieurs *driver* peuvent détourner les mêmes appels système sans pour autant nuire à la sécurité du système ou à sa stabilité. Le second avantage de cette méthode est la possibilité de réaliser des contrôles en post-traitement mais aussi en prétraitement. En effet, lorsqu'une fonction est détournée de la table des appels système, elle est exécutée directement par le *driver* si l'exécution est autorisée. Mais dans ce cas, c'est aussi le *driver* qui récupère le retour de la fonction. Il peut ainsi réaliser un nouveau contrôle.

Cette méthode de détournement des appels système permet de contrôler toutes les actions sur le système réalisées par des applications mais pas les *driver* qui résident en espace noyau et qui n'utilisent pas cette table pour réaliser des actions sur le système.

1. `rustock : https://www.securelist.com/en/analysis/204792011/Rustock_and_All_That`

Limites et légitimité Cette technique modifie la mémoire du noyau, c'est-à-dire que les modifications ne sont pas pérennes et ne résistent pas à un redémarrage. Il est donc nécessaire de recommencer l'opération de détournement à chaque démarrage du système. Cette technique a été énormément utilisée par les logiciels de sécurité comme les antivirus ou les pare-feux sur les systèmes Windows XP. Même si cette technique a été utilisée par les éditeurs de logiciels de sécurité, elle n'est pas recommandée par Microsoft. En effet, une mauvaise gestion des détournements entraîne la génération de BSOD.

De plus, pour contrôler toutes les actions sur le système, il est nécessaire de détourner un par un tous les appels système, opération qui peut se révéler très longue. Par exemple, sur un Windows 7, on trouve plus de 400 appels système alors qu'il n'y en a que 200 sur Windows XP. Cela nous amène à une première limite de cette technique : la portabilité entre les différentes versions de Windows. A chaque nouvelle version de Windows, la table des appels système est modifiée, il faut donc refaire le détournement pour chaque nouveau Windows. De plus, le traitement de chaque fonction peut être différent entre les versions de Windows puisque les paramètres des fonctions ne sont pas toujours les mêmes.

Certains logiciels de sécurité détournent certains appels système dans le but de protéger leur application. Ce détournement a pour but de contrôler la terminaison des processus et ainsi protéger leurs processus et autres services d'un `kill` provenant d'un processus non autorisé à le faire. Il existe néanmoins des techniques contournant les protections [2] mises en place au niveau de la table des appels système. Il devient ainsi possible de tuer les processus protégés par ces détournements.

Enfin, depuis Windows Vista en version 64 bits et sur les versions supérieures de Windows, la table des appels système est protégée par le mécanisme appelé *Kernel Patch Protection* [3]. Cette protection empêche tout *driver* de modifier cette table. Donc la technique de détournement de la table des appels système n'est pas portable sur les systèmes Windows 64 bits supérieurs à Windows Vista. Il faut noter que certaines personnes ont pu réaliser ces modifications mais en désactivant le *Kernel Patch Protection*, ce qui revient à désactiver un mécanisme de sécurité essentiel aux nouvelles versions de Windows.

Pour que le contrôle réalisé par l'observateur soit efficace, il est nécessaire de contrôler le chargement de tous les *driver*, puisqu'un *driver* pourrait supprimer ces détournements, et de vérifier constamment que :

— la table des appels système possède toujours les modifications faites par le *driver* ;
— qu'aucun *driver* malveillant ne soit chargé.

C.2 *Inline Hook*

Le seconde technique pour détourner les appels système est appelée *inline hook*. A la différence de la modification de la table des appels système qui n'est réalisable qu'en espace noyau, l'*inline hook* est une technique qui s'applique à n'importe quelle fonction et peut être réalisée en espace utilisateur tout comme en espace noyau.

Avantages et portée Grâce à l'utilisation des frameworks, *Microsoft Research* ou *easyhook*, le travail de détournement est grandement facilité et la portabilité entre les différentes versions de Windows est assurée.

Depuis les systèmes Windows XP, le *junk code* des fonctions de l'*API Win32* a été normalisé par Microsoft. Ainsi, il n'est plus nécessaire de "rechercher" l'emplacement libre en début de fonction pour placer le saut inconditionnel.

2. `http://www.kernelmode.info/forum/viewtopic.php?f=11&t=1878`
3. `http://blogs.msdn.com/b/windowsvistasecurity/archive/2006/08/11/695993.aspx`

En modifiant toutes les fonctions du noyau, il est possible de contrôler toute l'activité du système, même des *driver* qui résident en espace noyau, à condition qu'ils utilisent les fonctions du noyau.

Limites et légitimité Le fait d'introduire un *junk code* en début de fonction est une réalisation de Microsoft pour justement permettre le *hot-patching*. On peut donc considérer que ces techniques sont autorisées par Microsoft, qui fournit à la fois l'espace nécessaire pour placer un saut inconditionnel mais aussi un *framework* facilitant le déploiement de cette technique.

Pour pouvoir modifier les fonctions du noyau, il est nécessaire de désactiver *Kernel Patch Protection* car il vérifie aussi l'intégrité des fonctions. En désactivant ce mécanisme de sécurité, la signature des *driver* qui sont chargés ne sera plus vérifiée, ce qui peut conduire au chargement de *driver* malveillant.

De plus, on se retrouve avec les mêmes inconvénients qu'avec la technique modifiant la table des appels système. Il faut en effet modifier chaque fonction pour contrôler de manière globale le système. Cette méthode ne se limitant pas qu'aux appels système cette technique devient extrêmement contraignante à mettre en place.

Enfin, la modification des fonctions implique de connaître parfaitement leur fonctionnement. Or, certaines fonctions ne sont pas documentées et peuvent être modifiées par Microsoft, que ce soit dans leur prototypage ou dans leur fonctionnement.

C.3 *Filter-driver*

La troisième technique que nous décrivons se base sur les *filter-driver*. Elle se base sur une architecture spécifique mise en place par Microsoft pour réaliser des contrôles sur les entrées/sorties au niveau du système de fichiers.

Avantages et portée Les *filter-driver* reposent sur un système entièrement normalisé et documenté par Microsoft. En plus de fonctionner sur toutes les architectures classiques, Microsoft s'engage à assurer la portabilité des fonctions de l'API qu'il propose. Ainsi, un *filter-driver* fonctionnant sur Windows XP fonctionne aussi sur Windows 7 ainsi que sur les différentes architectures 32 et 64 bits.

Un *filter-driver* ne gère que les accès au système de fichiers, mais grâce à l'utilisation des IRP, qui regroupent des classes d'action, il est possible de contrôler non seulement les applications mais aussi les autres *driver* qui réalisent des opérations sur le système de fichiers. Il est ainsi possible de contrôler l'ensemble du système sans pour autant devoir modifier toutes les fonctions.

Limites et légitimité Comme c'est un mécanisme entièrement documenté et décrit par Microsoft, c'est le système qu'il faut utiliser pour détourner le flux d'exécution du système. Comme les appels système sont contenus dans les IRP, on ne détourne plus directement les appels système, mais une classe d'appel système.

Comme nous l'avons expliqué, le modèle des *filter-driver* repose sur des *couches* de *driver*. Ils sont représentés comme un empilement de *driver*. Cela signifie que potentiellement, le *driver* précédent dans l'opération, que ce soit en prétraitement ou en post-traitement, peut modifier les informations de l'IRP sans que le *filter-driver* ne puisse le savoir. Il faut donc contrôler les *driver* autorisés à se charger dans la couche de *driver*.

Damien GROS

Protection obligatoire répartie

La thèse porte sur deux enjeux importants de sécurité. Le premier concerne l'amélioration de la sécurité des systèmes Linux présents dans le calcul intensif et le second la protection des postes de travail Windows. Elle propose une méthode commune pour l'observation des appels système et la répartition d'observateurs afin de renforcer la sécurité et mesurer les performances obtenues. Elle vise des observateurs du type moniteur de référence afin de garantir de la confidentialité et de l'intégrité. Une solution utilisant une méthode de calcul intensif est mise en œuvre pour réduire les surcoûts de communication entre les deux moniteurs de référence SELinux et PIGA. L'évaluation des performances montre les surcoûts engendrés par les moniteurs répartis et analyse la faisabilité pour les différents noeuds d'environnements de calcul intensif. Concernant la sécurité des postes de travail, un moniteur de référence est proposé pour Windows. Il repose sur les meilleures protections obligatoires issues des systèmes Linux et simplifie l'administration. Nous présentons une utilisation de ce nouveau moniteur pour analyser le fonctionnement de logiciels malveillants. L'analyse permet une protection avancée qui contrôle l'ensemble du scénario d'attaque de façon optimiste. Ainsi, la sécurité est renforcée sans nuire aux activités légitimes.

Mots clés : sécurité, contrôle d'accès obligatoire, systèmes d'exploitation, logiciels malveillants, poste de travail, calcul intensif

Distributed mandatory protection

This thesis deals with two major issues in the computer security field. The first is enhancing the security of Linux systems for scientific computation, the second is the protection of Windows workstations. In order to strengthen the security and measure the performances, we offer a common method for the distributed observation of system calls. It relies on reference monitors to ensure confidentiality and integrity. Our solution uses specific high performance computing technologies to lower the communication latencies between the SELinux and PIGA monitors. Benchmarks study the integration of these distributed monitors in the scientific computation. Regarding workstation security, we propose a new reference monitor implementing state of the art protection models from Linux and simplifying administration. We present how to use our monitor to analyze the behavior of malware. This analysis enables an advanced protection to prevent attack scenarii in an optimistic manner. Thus, security is enforced while allowing legitimate activities.

Keywords : security, mandatory access control, operating systems, malware, workstation, high performance computing

Laboratoire d'Informatique
Fondamentale d'Orléans
Bâtiment IIIA Rue Léonard de Vinci
B.P. 6759
F-45067 ORLEANS Cedex 2

CEA/DAM/DIF
Bruyères-le-Châtel
91297 Arpajon Cedex