n° d'ordre : **2985**

Thèse présentée devant

l'Université Rennes I

pour obtenir le titre de

Docteur de l'Université de Rennes I

mention Informatique

par

Jakub Zimmermann

Équipe d'accueil : Sécurité des Systèmes Informatiques et Réseaux
École Doctorale : MATISSE
Composante universitaire : IFSIC

Détection d'Intrusions Paramétrée par la Politique par Contrôle de Flux de Références

Thèse soutenue le **16 décembre 2003** devant la Commission d'examen :
- Jean-Marc Jezequel, Président du Jury
- Yves Deswarte, Rapporteur
- Hervé Debar, Rapporteur
- Gerardo Rubino, Directeur de Thèse
- Frédéric Cuppens, Examinateur
- Ludovic Mé, Examinateur

Détection d'intrusions paramétrée par la politique par contrôle de flux de références

8 mars 2004

Je comprends comment, je ne comprends pas pourquoi.

- George Orwell, *1984*

Remerciements

Je souhaite remercier en particulier les personnes suivantes, qui m'ont permis de réaliser cette thèse :

- M. Gerardo Rubino, directeur de la thèse ;
- Le campus Supélec Rennes pour avoir financé cette thèse et m'avoir accueilli dans ses locaux ;
- MM. Hervé Debar et Yves Deswarte, pour avoir accepté d'évaluer mes travaux ;
- Le Président du Jury, M. Jean-Marc Jezequel ;
- Les autres membres du Jury, MM. Ludovic Mé et Frédéric Cuppens ;
- M. Ludovic Mé, pour m'avoir accueilli dans son équipe et pour son excellente direction de mes travaux ;
- Les membres de l'équipe SSIR, en particulier MM. Christophe Bidan et Eric Totel, pour leur aide, leurs conseils et nos discussions enrichissantes ;
- Ma mère, ma soeur et mon frère, pour leurs encouragements et leur soutien sans faille au cours de ces trois années ;
- Les enseignants-chercheurs et thésards de Supélec Rennes, pour leur sympathie et le cadre de travail très agréable qu'ils m'ont offert.

Table des matières

1 Introduction **13**

2 Travaux antérieurs **17**
2.1 Détection d'intrusions classique 17
2.1.1 Détection d'anomalies 18
2.1.1.1 Construction du profil 18
2.1.1.2 Détection des déviations 20
2.1.1.3 Fiabilité et pertinence 20
2.1.2 Détection par scénario 20
2.1.2.1 Construction de la base 21
2.1.2.2 Détection des scénarios 21
2.1.2.3 Fiabilité et pertinence 22
2.1.3 Limites et perspectives 22
2.2 Détection paramétrée par la politique 23
2.2.1 Détection à base de spécification de comportement . . 23
2.2.1.1 Définition de la politique 24
2.2.1.2 Contrôle de la politique 24
2.2.1.3 Discussion 25
2.2.2 Contrôle de non-interférence 26
2.2.2.1 Définition des politiques de sécurité 26
2.2.2.2 Contrôle de la politique 27
2.2.2.3 Utilisation en détection d'intrusions 28
2.2.2.4 Discussion 29
2.3 Contrôle d'accès 30
2.3.1 Modèle général de type *HRU* 31
2.3.1.1 Transfert de privilèges 32
2.3.1.2 Divulgation d'information 33
2.3.1.3 Garanties de sécurité 34
2.3.2 Contrôle d'accès mandataire 34
2.3.2.1 Modèle « Bell & LaPadula » 35
2.3.2.2 Modèle « Biba » 36
2.3.2.3 Muraille de Chine 36
2.3.3 Le modèle « Take-Grant » 37

2.3.3.1 Modélisation des politiques de sécurité 37
2.3.3.2 Divulgation d'informations 38
2.3.3.3 Validation des politiques 39
2.3.3.4 Graphe d'action et conspiration 39
2.3.3.5 Discussion 40
2.4 Contrôle des flux d'informations 40
2.4.1 Treillis de Denning 40
2.4.1.1 Représentation du système 40
2.4.1.2 Contrôle des flux 41
2.4.2 Non-interférence 42
2.4.3 Modèles pour environnements spécifiques 43
2.4.4 Confinement de processus (DTE) 43
2.4.4.1 Définition des politiques de sécurité 44
2.4.4.2 Modèle de contrôle d'accès 44
2.4.4.3 Contrôle de l'exécution 46
2.4.4.4 Discussion 47
2.5 Résumé 48

3 Modèle du système 51
3.1 Attaques par délégation 51
3.1.1 Violations de politique de sécurité 51
3.1.2 Exemple 52
3.1.2.1 Attaque contre *lpd* 52
3.1.2.2 Attaque contre *OpenSSH* 53
3.1.2.3 Attaque contre *Apache* 54
3.1.3 Attaques et flux d'information 55
3.2 Représentation du système 56
3.3 Opérations et flux d'informations 58
3.3.1 Opérations élémentaires 58
3.3.2 Dépendance causale 59
3.4 Politique de sécurité et domaines 60
3.4.1 Interprétation de la politique de sécurité 60
3.4.1.1 Exemple 1 : contrôle d'accès discrétionnaire . 60
3.4.1.2 Exemple 2 : muraille de Chine 61
3.4.2 Domaines 62
3.5 Modèle formel 63
3.5.1 Références 63
3.5.2 Légalité des opérations et règle d'unicité 64
3.5.3 Flux de références 65
3.6 Détection d'intrusions 69
3.7 Résumé 70

4 Implémentation 71
4.1 Modèle du système Linux 71
4.1.1 Objets et méthodes 72
4.1.1.1 Fichiers et descripteurs d'E/S 72
4.1.1.2 Attributs des fichiers 73
4.1.1.3 Sockets 74
4.1.1.4 Mémoire virtuelle 74
4.1.1.5 Messages entre processus 75
4.1.1.6 Périphériques et variables du noyau 75
4.1.2 Fichiers non-atomiques 76
4.1.2.1 Contenu non-atomique 76
4.1.2.2 Solutions potentielles 76
4.1.3 Opérations 77
4.2 Politiques de sécurité 78
4.2.1 Droits d'accès 78
4.2.2 Objets virtuels et authentification 80
4.2.3 Identité des sujets 81
4.2.4 Authentification 81
4.3 Réalisation 82
4.3.1 Génération des traces 82
4.3.2 Génération de l'état initial 83
4.3.3 Contrôle des flux 84
4.4 Retour d'expérience 84
4.5 Application à d'autres systèmes d'exploitation 85

5 Utilisation et tests 89
5.1 Tests en environnement contrôlé 89
5.1.1 Environnement de tests 89
5.1.2 Politiques de sécurité 90
5.1.3 Description des attaques 91
5.1.3.1 URL illégales 91
5.1.3.2 Exécution de programmes *cgi-bin* 92
5.1.3.3 Débordements de tampon 93
5.1.3.4 Attaques par fenêtre temporelle (*Race-conditions*) 93
5.1.4 Opérations locales exécutées 94
5.1.5 Tests et résultats 95
5.1.6 Discussion 97
5.2 Tests en utilisation réelle 98
5.2.1 Environnement de tests 98
5.2.2 Politique de sécurité 99
5.2.2.1 Service Web 99
5.2.2.2 Services de courrier électronique 100
5.2.3 Description des attaques 101
5.2.3.1 Erreur de format dans Exim (CVE-2001-0690) 101

5.2.3.2 Erreur de format dans rpc.statd (CVE-2000-0666) . . . 102
5.2.3.3 *Race-condition* dans le noyau Linux (CAN-2003-0127) . . . 102
5.2.3.4 Autres . . . 103
5.2.4 Tests et résultats . . . 103
5.2.4.1 Utilisation des « exploits » . . . 103
5.2.4.2 Détections correctes . . . 104
5.2.4.3 Faux négatifs . . . 105
5.2.4.4 Faux positifs . . . 105
5.2.5 Discussion . . . 106

6 Conclusion **109**

Bibliographie **113**

A Langage de configuration **119**

B Flux de références et non-interférence **123**
B.1 Sujets et données . . . 123
B.2 Politique de non-interférence . . . 124

Liste des tableaux

2.1 Profil probabiliste de séquence d'événements 19

3.1 Vulnérabilités vs. violations de politique 53

4.1 Objets et méthodes du système Linux 72
4.2 Appels système pris en compte 87
4.3 Surcharge mémoire causée par le détecteur 87
4.4 Compilation complète d'un noyau Linux 88

5.1 Fichiers utilisés lors des attaques 90
5.2 Tests avec *BuggyHTTP* 96
5.3 Systèmes testés 98
5.4 Résumé des résultats des tests 104

Table des figures

2.1 Attaque contre *lpr* 29

3.1 Exemple le décomposition de flux 60
3.2 Interprétation du contrôle d'accès 62
3.3 Domaines correspondant à la politique 63
3.4 Résultat du flux de o_1 vers o_2 68

4.1 Redirection des appels système 82
4.2 Exemple de déclaration d'objets virtuels 83

5.1 Flux autorisés par le contrôle d'accès 91
5.2 Flux autorisés par la politique affinée 92
5.3 Politique de sécurité définie manuellement sur les systèmes 1 et 2 99

Chapitre 1

Introduction

Avec l'avènement de la « société de l'information », l'informatique joue désormais un rôle central dans la vie économique et sociale. Dans ces conditions, la sécurité des systèmes informatiques revêt une importance primordiale aussi bien à l'échelle de l'individu ou de l'entreprise qu'à celle de l'État ou des institutions mondiales.

Afin d'assurer la qualité de service requise, il est donc nécessaire de définir, pour chaque système informatique, une *politique de sécurité* qui réglemente l'accès à l'information contenue dans le système et aux services de ce dernier. Cette politique s'exprime par des règles fixant trois objectifs distincts :

confidentialité : pour chaque information est défini l'ensemble des utilisateurs autorisés à y accéder ;

intégrité : pour chaque information est défini l'ensemble des utilisateurs autorisés à la modifier, l'ensemble des utilisateurs qui ne sont pas autorisés à en empêcher la modification et, éventuellement, un ensemble de valeurs autorisées pour cette information ;

disponibilité : les services du système doivent être opérationnels et accessibles selon des modalités définies.

En soi, la définition d'une politique de sécurité ne garantit pas seule le fonctionnement correct du système. Une politique peut en effet être violée, malencontreusement (par exemple, à la suite d'une erreur de conception logicielle), ou délibérément par un utilisateur hostile. Il est donc nécessaire de s'assurer du respect de cette politique et d'en détecter les violations.

Les techniques et outils disponibles aujourd'hui pour faire appliquer une politique de sécurité peuvent être classés dans deux catégories : les outils de *prévention*, dont le but premier est la mise en oeuvre d'une politique, et les outils de *détection* destinés à signaler les cas de violation de celle-ci. De manière générale, la politique de sécurité à proprement parler relève souvent des seuls outils du type préventif. Cependant, l'expérience montre qu'une approche basée uniquement sur la prévention est insuffisante.

Les outils de prévention possèdent des limites ou des erreurs de conception que l'attaquant pourra éventuellement exploiter. Les outils disponibles pour la détection s'avèrent quant à eux souvent peu fiables, car basés souvent sur des données *empiriques* (par exemple, des signatures d'attaque) ou des paramètres relatifs aux *symptômes* d'attaques et non à leurs effets. Or, ces données reflètent principalement la connaissance que l'on possède du système et/ou des attaques potentielles, mais elles n'expriment pas une politique de sécurité. Une alerte levée par l'outil de détection ne correspond donc pas systématiquement à une violation effective de la politique de sécurité, mais seulement à la présence d'un *symptôme*, associé empiriquement au *risque de violation* de la politique. Par exemple, un comportement anormal ou un comportement typique d'une intrusion peut effectivement conduire à une violation de la politique, mais il peut également s'avérer bénin, voire inoffensif sur le système informatique considéré.

De notre point de vue, le déficit de sécurité des systèmes informatiques actuels est dû à un ensemble de facteurs dont :

- une prise en compte insuffisante de l'aspect *sécurité* dans la conception des systèmes informatiques courants (existence de nombreuses failles dans les logiciels) ;
- l'imperfection des outils de prévention utilisés pour implémenter une politique, ce qui permet sa violation ;
- la non-adéquation des outils de détection à la politique de sécurité mise en oeuvre ;
- l'absence, la plupart du temps, d'une étude théorique du problème de la détection des violations d'une politique de sécurité [1] ;
- une information insuffisante des utilisateurs, qui ont tendance à négliger des règles de sécurité, même élémentaires.

Ce constat motive la recherche dans le domaine de la détection d'intrusions *paramétrée par la politique* : l'objectif de ces travaux consiste à concevoir des outils de détection basés uniquement sur une définition *logique* de la politique de sécurité, en excluant toute donnée empirique. En effet, un détecteur efficace et fiable doit être en mesure de signaler toute violation de la politique de sécurité, y compris par des moyens nouveaux ou inconnus. Or, ceci requiert que la détection est basée sur d'autres critères que la connaissance des symptômes des attaques connues. Simultanément, les alertes produites par le détecteur doivent idéalement toutes correspondre à des violations effectives de la politique, c'est-à-dire qu'il ne doit pas y avoir de « fausse alerte ».

Notre projet s'inscrit dans cette voie. Nous défendons dans cette thèse la position suivante :

1. un outil efficace de détection d'intrusions doit être basé sur une définition de la politique de sécurité ;

2. le système de détection doit être implémenté au niveau *système* (*host-*

based intrusion detection), et non au niveau *réseau* (*network-based intrusion detection*). En effet, une politique sécurité est généralement définie de manière *locale*. D'autre part, nous pensons que les nouveaux services et les moyens de protection des communications déployés (chiffrement, garantie de débit, mobilité, etc.) pourraient rendre la détection par analyse du trafic difficile, voire impossible à terme.

A l'appui de cette position, nous proposons dans cette thèse une approche de détection d'intrusions paramétrée par la politique. Pour cela, nous avons retenu la démarche suivante :

- nous nous limitons aux aspects *confidentialité* et *intégrité* d'une politique de sécurité, l'aspect disponibilité étant un problème très différent exigeant d'autres approches ;
- nous représentons, à l'aide d'un formalisme unique, le système d'exploitation considéré et la politique de confidentialité/intégrité mise en oeuvre, afin de définir, sous forme d'un prédicat logique, la notion de violation. Naturellement, ce modèle de la violation ne sera effectivement utilisable que s'il couvre un nombre suffisant d'attaques réelles *différentes* ;
- nous *écartons* une représentation explicite des comportements conformes à la politique de sécurité. Le modèle ne sera effectivement utile que s'il est applicable à des systèmes réalistes, dont il n'est pas possible de caractériser le comportement « attendu » (pour des raisons de complexité et/ou à cause du fait que les connaissances du fonctionnement précis des programmes ne sont pas disponibles) ;
- nous en dérivons un algorithme de détection des violations de politique décrites par ce modèle. Nous souhaitons que cet algorithme soit paramétré *uniquement* par le modèle du système et la politique de confidentialité/intégrité ;
- afin d'en évaluer les résultats, nous implémentons ce modèle sur un système d'exploitation réaliste.

Ce mémoire est organisé de la manière suivante. Dans le chapitre 2, nous débutons par une présentation des travaux effectués dans le domaine de la définition des politiques de sécurité et de la détection d'intrusions, puis nous décrivons dans le chapitre 3 le modèle théorique du système, de la politique et de ses violations que nous proposons. Enfin, nous traitons dans le chapitre 4 de sa mise en oeuvre sur un système réel et nous terminons, dans le chapitre 5, par une discussion des résultats expérimentaux obtenus.

Chapitre 2

Travaux antérieurs

Ce chapitre présente les travaux menés dans les domaines liés à cette thèse. Nous utiliserons le terme *intrusion* pour désigner une *violation de politique de sécurité*. Cette politique consiste en un ensemble de propriétés que le système doit respecter et un ensemble de règles qui définissent les modifications possibles de l'état de protection [2]. Une intrusion correspond donc soit à une violation des propriétés (exécution d'une action contraire à ces propriétés), soit à une modification des propriétés en violant les règles.

Nous nous intéressons aux mécanismes utilisés pour détecter ces violations : avant tout, la détection d'intrusions classique, dont nous décrivons les limites, et les travaux dans le domaine de la détection d'intrusions paramétrée par la politique à base de spécification de comportement et de contrôle de non-interférence. Bien qu'offrant des résultats intéressants, ces approches souffrent néanmoins selon nous de problèmes intrinsèques. Pour proposer une solution à ces problèmes, nous souhaitons adopter la définition d'une politique de sécurité comme point de départ. La fin du chapitre est donc consacrée aux modèles de contrôle d'accès et de contrôle de flux d'informations, qui demeurent le principal moyen de mettre en oeuvre une politique de confidentialité ou d'intégrité.

2.1 Détection d'intrusions classique

Les méthodes classiques de détection d'intrusions peuvent être classées dans deux catégories. La détection d'*anomalies*[1] exploite la définition empirique d'un fonctionnement « sûr » et cherche à détecter des déviations par rapport à cette norme. La détection *par scénario*[2] repose au contraire sur une connaissance préalable des scénarios d'attaque, dont les occurrences sont détectées.

[1] *anomaly detection* dans la littérature en langue anglaise.

[2] *misuse detection* ou *scenario-based detection.*

Les performances des techniques de détection d'intrusions s'expriment traditionnellement par :

la fiabilité (ou **couverture**) du détecteur : idéalement, toute intrusion doit effectivement donner lieu à une alerte. Une intrusion non signalée constitue une défaillance du détecteur, appelée couramment « faux négatif ». La fiabilité d'un détecteur est liée à son taux de « faux négatifs » (c'est-à-dire le pourcentage d'intrusions non détectées), qui doit être le plus bas possible ;

la pertinence (ou **crédibilité**) des alertes : idéalement, toute alerte doit correspondre à une intrusion effective, toute « fausse alerte » (également appelée « faux positif ») diminue naturellement la crédibilité. Un bon détecteur doit présenter un nombre de fausses alertes par unité de temps aussi bas que possible.

Nous verrons respectivement dans les sections 2.1.1 et 2.1.2 que les deux méthodes classiques souffrent soit d'un problème de fiabilité, soit de pertinence.

2.1.1 Détection d'anomalies

La notion de *détection d'intrusions* a été définie par Anderson en 1980 dans [3]. Constatant qu'une politique de sécurité, implémentée au moyen des mécanismes mis à disposition par le système peut être violée, l'auteur propose une approche consistant à observer le comportement des utilisateurs et le comparer à un comportement de référence, appelé *profil*. L'utilisation de cette méthode nécessite :

1. de construire un profil pertinent au vu d'une politique de sécurité ;
2. de définir les critères permettant d'évaluer la « déviation » du comportement observé par rapport à ce profil.

De manière générale, la détection d'anomalies présente plusieurs avantages. La définition du profil détermine le comportement attendu du système mais ne fait aucune hypothèse sur les « anomalies » qui s'en écartent. Toute forme de déviation par rapport au profil sera détectée et donnera lieu à une alerte : le système se montrera donc particulièrement fiable. Par ailleurs, un profil n'évolue que rarement, en cas de modification de la politique de sécurité ou d'évolution radicale du système informatique considéré. Un détecteur d'anomalies ne requiert donc théoriquement qu'une maintenance minimale.

2.1.1.1 Construction du profil

Par définition, le profil est constitué d'un ensemble de mesures empiriques correspondant à un comportement « normal ».

La construction du profil résulte donc d'une *phase d'apprentissage*, au cours de laquelle le système est soumis à une utilisation « type » dans le but

type d'événement	probabilité
démarrage de session utilisateur	75%
retour à l'invite *login*	20%
verrouillage du compte utilisateur	5%
autre	0%

TAB. 2.1 – Profil probabiliste de séquence d'événements

de mesurer les valeurs du profil caractéristique. Ces paramètres peuvent être de nature probabiliste ou statistique.

probabiliste : Étant donné une séquence d'événements, le profil indique la distribution probabiliste du prochain événement attendu. Par exemple, étant donné la séquence d'événements :

1. Entrée d'un nom d'utilisateur à l'invite[3] *login* ;
2. Entrée d'un mot de passe à l'invite *password*.

La table 2.1 indique l'événement suivant de la séquence.

statistique : Le profil indique l'évolution dans le temps de données telles que la charge processeur, l'occupation mémoire, l'activité des entrées / sorties etc.

Cet apprentissage pose des problèmes pratiques. En effet, aucune intrusion ne doit se produire au cours de cette phase. En pratique, cela impose souvent que le système en cours d'apprentissage soit totalement contrôlé et isolé des interactions avec l'environnement, afin de garantir que le comportement mesuré au cours de l'apprentissage ne soit pas biaisé. L'occurrence d'intrusions durant la phase d'apprentissage conduit naturellement à la création d'un profil reflétant ces intrusions. Un utilisateur hostile peut ainsi éventuellement entraîner délibérément un biais du profil, qu'il pourra exploiter pour attaquer le système sans être détecté, lorsque le détecteur d'anomalies sera opérationnel. Simultanément, le fonctionnement du système doit être suffisamment réaliste afin de donner lieu à un profil pertinent et caractéristique de l'utilisation effective du système.

Par ailleurs, la phase d'apprentissage pose intrinsèquement le problème de l' « étalonnage » du profil, c'est-à-dire de la précision requise des mesures. Afin d'obtenir une fiabilité de détection élevée, le profil doit être aussi représentatif que possible du comportement attendu. Cela introduit un risque de *sur-apprentissage* [4] : un profil trop précis conduit à considérer comme suspect *tout comportement différent de celui lors de l'apprentissage*. La pertinence des alertes produites est alors très basse, voire nulle.

[3] « *prompt* »

2.1.1.2 Détection des déviations

Le profil consiste en un ensemble de paramètres décrivant le comportement « normal ». Le détecteur mesure ces paramètres dans le comportement observé et compare les résultats aux valeurs attendues du profil. Une alerte est alors générée si les mesures observées ne correspondent pas au profil, par exemple :
- apparition trop fréquente d'un événement à probabilité très basse ;
- au contraire, non-apparition d'un événement attendu ;
- différence importante entre la valeur statistique attendue et celle observée ;
- etc.

La définition des « seuils de déclenchement », c'est-à-dire de la différence minimum entre la valeur attendue et la valeur observée donnant lieu à une alerte, est une difficulté majeure. Outre ce problème du « calibrage », il est en outre nécessaire de prendre en compte l'évolution des habitudes des utilisateurs et de l'utilisation du système informatique.

Les seuils de déclenchement liés au profil doivent donc être régulièrement ajustés. Or, si une attaque donne lieu à des mesures dépassant ces seuils, un attaquant peut procéder progressivement : adopter un comportement visant délibérément à ne dévier que très légèrement des valeurs du profil pendant une longue période, de manière à provoquer un ajustement des seuls. En poursuivant cette stratégie, il peut aboutir à une configuration du détecteur dans laquelle l'attaque originale apparaîtra conforme aux mesures attendues et ne sera pas détectée.

2.1.1.3 Fiabilité et pertinence

Par construction, un détecteur d'anomalies signale toute déviation par rapport au profil, c'est-à-dire qu'il offre théoriquement une *fiabilité maximale*. Néanmoins, toute déviation par rapport au profil ne correspond pas nécessairement à une alerte : par exemple, un utilisateur peut éprouver des besoins ponctuels tels que la compilation d'un programme qui engendre des mesures totalement atypiques pour son profil. Il s'agit d'utilisations légitimes, qui seront néanmoins signalées en tant que violations du profil. La détection d'anomalies présente donc une *pertinence médiocre* : souvent, plus de 90% des alertes produites s'avèrent être des faux positifs [5, 1].

2.1.2 Détection par scénario

Cette méthode de détection d'intrusions vise à détecter les occurrences d'attaques connues. Un détecteur d'intrusions par scénario comprend :
- un ensemble de sondes produisant un flux d'événements ;
- une base de « signatures », c'est-à-dire une base de données comprenant

des motifs dans le flux d'événements caractéristiques des scénarios d'intrusions connus ;
- un algorithme de recherche de motif, comparant le flux d'événements avec les signatures contenues dans la base.

Un détecteur par scénario exige une maintenance active : afin d'offrir une fiabilité satisfaisante, la base de signatures doit en permanence évoluer en fonction des connaissances des scénarios d'attaque.

2.1.2.1 Construction de la base

La construction de la base de signatures exige des connaissances précises des attaques dont elle contient les signatures. Ces informations peuvent être obtenues par exemple par veille technologique, par analyse de code source, etc.

De nombreux travaux portent sur le développement de langages de description d'attaques adaptés [?, 6, 7]. Un tel langage doit se montrer :
- expressif pour permettre une description à la fois simple, précise et concise d'un large spectre d'attaques ;
- facilement implémentable dans l'environnement considéré (système d'exploitation, codage des signatures, sondes, mode de reconnaissance de motifs).

2.1.2.2 Détection des scénarios

L'objectif d'un détecteur par scénario est de lever une alerte si un motif correspondant à une signature stockée dans la base est détecté dans le flux d'événements. De nombreuses approches ont été proposées.

L'algorithme le plus simple consiste en une simple reconnaissance de motifs par automate fini (*pattern matching*) : c'est l'analyse dite « mono-événement ». Dans ce cas, une signature correspond à une simple suite d'événements ou une expression régulière, utilisée pour filtrer le flux d'événements à la recherche des occurrences des attaques. Cette approche atteint vite ses limites, car il s'avère difficile de construire une signature assez générique pour couvrir différentes variantes possibles (éventuellement encore inconnues) d'un scénario, et simultanément assez précise pour discriminer des cas qui ne correspondent pas au scénario considéré.

Les limites du *pattern-matching* classique résident dans le fait que plusieurs scénarios différents peuvent donner lieu à une même séquence d'événements, en fonction de paramètres tiers. Ce problème est traité par des modèles de corrélation « multi-événements », intégrant des pré-, post-conditions et des assertions [6, 8], des algorithmes de recherche d'occurrences autres que le simple *pattern-matching* : par exemple, les algorithmes génétiques [9] ou les réseaux bayésiens [10], ou encore des approches intégrant l'analyse de la configuration du système [11].

2.1.2.3 Fiabilité et pertinence

Un détecteur de scénario ne lève d'alerte que si le flux d'événements correspond effectivement à une des signatures de la base. La fiabilité de la détection dépend donc directement de la qualité de cette base, plus précisément de sa richesse, de son actualité et de la nature suffisamment générique des signatures.

En principe, la pertinence de cette méthode s'avère excellente : un « faux positif » n'est généré que si le flux d'événements correspond à l'une des signatures *alors que l'intrusion correspondante n'a pas réellement lieu.* Toutefois, l'exigence que les signatures soient génériques peut entraîner la construction de signatures « trop génériques », correspondant à un vase spectre de motifs dans le flux d'événements, et qui couvrent aussi des cas bénins.

2.1.3 Limites et perspectives

Si la détection d'anomalies offre une excellente fiabilité, elle souffre d'une faible pertinence. Par ailleurs, la phase d'apprentissage et le mécanisme de détection posent intrinsèquement le problème du biais des mesures et/ou du sur-apprentissage. La cause peut en être une manipulation volontaire (par exemple, un attaquant cherchant à déformer le profil à son avantage) ou un mauvais calibrage, par exemple en cas de sur-apprentissage. L'exploitation pratique de cette méthode nécessite donc une prise en compte de ce problème. En particulier, une étude de l'évolution du profil ou des seuils dans le temps peut être nécessaire pour détecter des tentatives de biaiser le système [12]. Une voie consiste à s'écarter progressivement d'un profil purement empirique, pour exploiter des connaissances du fonctionnement du système et définir des profils par *description logique du comportement.* Comme nous le verrons plus loin, une solution consiste à spécifier le comportement *attendu* du système [13]. Une autre approche consiste à déterminer des « points faibles », c'est-à-dire des séquences courtes d'événements d'opérations exécutées, rendant possible une violation de la politique [14]. Dans [15], Forrest, Hofmeyr et Somayaji proposent une méthodologie permettant d'observer les séquences d'opérations (c'est-à-dire d'appels systèmes) exécutées lors d'un fonctionnement sain et en extraire des motifs caractéristiques et reproductibles. Lors de la détection, ces motifs sont recherchés dans les séquences d'appels systèmes observées. Cette approche se montre peu coûteuse et efficace pour détecter des détournements de programmes privilégiés, néanmoins, elle nécessite toujours une phase d'apprentissage et détecte des violations *potentielles* de la politique de sécurité, et non des violations *effectives.*

La détection par scénario assure théoriquement une pertinence plus élevée, mais une fiabilité satisfaisante nécessite une maintenance active de la base de signatures. La construction de la base se heurte par ailleurs au fait qu'afin

d'assurer une fiabilité de détection suffisante et des performances acceptables, les signatures se montrent trop génériques, ce qui conduit à un taux élevé de fausses alertes.

Chacune de ces deux approches est donc considérée insuffisante en tant que telle. Afin d'améliorer la fiabilité et la pertinence de la détection, de nombreux travaux étudient les possibilités de *corrélation*, en particulier :
- comparaison des résultats produits par différents détecteurs basés sur différentes techniques ;
- prise en compte des connaissances du système et de son environnement, afin d'éliminer des alertes correspondant à des scénarios impossibles ou bénins sur le système-cible [16] ;
- corrélation des événements observés ;
- etc.

Le principal inconvénient lié à la corrélation réside dans la complexité des modèles et l'importance des moyens nécessaires pour sa mise en oeuvre.

2.2 Détection paramétrée par la politique

Selon un constat largement partagé, les limites des méthodes traditionnelles proviennent essentiellement du fait que celles-ci n'intègrent pas la notion de *politique de sécurité*. De fait, ces approches ne détectent pas des violations de politique strictement parlant, mais plutôt des symptômes de violations potentielles. *A contrario,* un détecteur d'intrusions paramétré par la politique ne devrait idéalement reposer sur aucune donnée empirique, mais uniquement sur la définition de la politique de sécurité. Dans la mesure où une telle approche repose elle aussi sur une analyse de comportement observé sous forme de traces, il en résulte que toute politique de sécurité définit implicitement un langage de traces. Une trace valide dans ce langage correspond alors à un comportement légal, tandis que toute trace n'appartenant pas au langage résulte d'une violation de la politique. Ainsi, un détecteur d'intrusions paramétré par la politique idéal détecte des violations *effectives* de la politique. Les réalisations actuelles s'approchent plus ou moins de cet objectif : il s'agit de la détection à base de spécification de comportement et du contrôle de non-interférence. Elles sont présentées dans la suite de cette section.

2.2.1 Détection à base de spécification de comportement

Cette approche a été initialement proposée par Ko [17]. Son principe consiste à spécifier explicitement le langage de traces. Il s'agit donc encore de comparer l'exécution des programmes à une norme. Cependant, cette norme n'est pas empirique et ne résulte pas d'un apprentissage. Au contraire, elle spécifie le comportement que le programme observé *devrait* avoir pour être considéré comme « sûr ». Nous pouvons donc dire qu'il s'agit d'une méthode

de détection à base de politique, où la politique réglemente le *comportement* des programmes. C'est en réalité une forme de détection d'anomalies.

2.2.1.1 Définition de la politique

Dans cette approche, une politique consiste directement en une spécification du comportement attendu. Cette spécification est utilisée pour discriminer les traces d'exécution (c'est-à-dire les séquences d'événements observés) et décider si la trace examinée est légale ou pas. Dans les réalisations actuelles, les traces sont des séquences d'appels système exécutés par le processus observé.

Chaque implémentation propose donc un *langage de spécification*. Selon les implémentations, la spécification se fait à l'aide d'un ensemble d'expressions régulières [18], d'une grammaire [17, 19] etc. A l'aide de ce langage, l'administrateur du système construit une grammaire des traces légales. Chaque règle terminale de la grammaire ainsi définie correspond à un appel système ; des implémentations avancées [20] proposent des macro-définitions telles que :

- *WriteOperations* : ensemble des appels système entraînant la création ou la modification des fichiers ;
- *ProcessInterference* : ensemble des appels système grâce auxquels un processus peut contrôler l'exécution d'un autre processus ;
- etc.

L'expressivité du langage de spécification est par ailleurs souvent renforcée par la possibilité d'associer aux règles des clauses qui affectent des variables d'état, et des prédicats portant sur la valeur des paramètres de l'appel système et/ou celle des variables d'état.

2.2.1.2 Contrôle de la politique

Le modèle de reconnaissance nécessaire pour valider les traces d'exécution peut varier d'un simple automate à états fini à une machine de Turing, en fonction du langage proposé pour la spécification. En pratique, la conception d'un détecteur à base de spécification privilégie soit l'expressivité du modèle, soit les performances de l'implémentation.

Dans tous les cas, le système de détection lui-même se résume à trois éléments :

1. Un *compilateur du langage de spécification*, produisant un module de reconnaissance ;
2. Un *système d'audit* enregistrant les appels système et renvoyant les traces d'exécution ;
3. Un *gestionnaire*, chargé de collecter les traces d'exécution des programmes soumis à la politique, leur appliquer les modules de reconnais-

sance associés et réagir en cas d'échec de la reconnaissance (c'est-à-dire en présence d'une trace non conforme à la spécification).

Dans une optique de détection pure, la réaction à un échec de reconnaissance se résume à la génération d'une alerte. Cependant, la méthode peut également être utilisée dans une optique *préventive*, visant à interdire les exécutions non conformes à la spécification.

La solution de prévention la plus simple consiste à interdire l'exécution de tout appel système qui viole la spécification. Une autre approche est proposée dans [13] : lorsque la politique de comportement est violée, le processus fautif est isolé du reste du système, de sorte que l'attaquant ayant détourné ce processus ne puisse pas corrompre davantage ce dernier. Dans l'implémentation actuelle, cela consiste à :

- modifier la politique relative à ce programme et interdire nombre d'opérations « sensibles » (par exemple, empêcher le processus d'ouvrir des sockets réseau) ;
- affecter une nouvelle identité de sujet (utilisateur + groupe) au processus, avec des droits très réduits.

2.2.1.3 Discussion

Cette approche étant une forme de détection d'anomalies, elle en conserve les avantages : aucune connaissance préalable des attaques n'est requise. De plus, la méthode s'avère facile à mettre en oeuvre et ouvre des perspectives intéressantes en tant que moyen de prévention ou de tolérance aux intrusions.

Dans [20], les auteurs présentent les résultats de leurs expériences, conduites sur un ensemble réduit de données (30 occurrences d'attaques). Le système de détection utilisait à la fois des spécifications génériques (par exemple, les programmes privilégiés ne doivent accéder qu'à un ensemble réduit de fichiers), des spécifications propres aux applications testées (serveur *ftp*, *http* etc...), et des spécifications particulières liées à la politique de sécurité du site (par exemple, les fichiers du répertoire *secret* ne doivent pas être accessibles via les utilitaires *cat* ou *cp*). Dans ces conditions, toutes les attaques reposant sur le détournement de programmes privilégiés, soit 82% de l'ensemble des attaques, ont été détectées avec succès avec 0% de fausses alertes.

Par ailleurs, certaines attaques restent indétectables avec cette méthode : celles liées à l'existence de *flux d'informations illégaux*, alors qu'un comportement identique mais accédant à une autre information serait légal (dans les exemples testés, il s'agissait de l'attaque par *HTTPTunnel* et de la tentative de se connecter en tant que *guest*). Pour ces attaques, l'écriture d'une spécification du comportement correct s'avère impossible et les auteurs ont dû au contraire écrire une spécification de comportement correspondant à l'attaque, c'est-à-dire utiliser une approche à base de scénario.

En conclusion, la détection à base de spécification de comportement se montre efficace pour détecter certains types d'attaques (l'abus de programmes privilégiés), à condition de posséder une connaissance préalable du fonctionnement de ces programmes. En pratique, la plupart des programmes privilégiés (ou « critiques », tels un serveur *http*) ont des comportements simples et bien connus, ce qui facilite l'écriture de ces spécifications. Le recours à la détection à base de scénario reste néanmoins nécessaire dans les cas où le seul comportement du programme ne permet pas de distinguer l'attaque d'un fonctionnement légitime. En utilisant le même langage de spécification pour modéliser les scénarios d'attaque, ces deux approches s'intègrent élégamment dans un même système de détection.

2.2.2 Contrôle de non-interférence

Dans [21], Ko et Redmond ont présenté une approche de la détection d'intrusions à base de politique où le langage de traces n'est pas défini explicitement, mais déduit d'une politique de sécurité donnée sous forme de propriété. Grâce à la méthode proposée, il est possible de détecter des violations de la politique mise en oeuvre utilisant des scénarios d'attaque connus ou inconnus. Cette étude constitue par ailleurs l'un des premiers travaux dans le domaine de la détection d'intrusions bâti sur une base théorique et non empirique, dans laquelle le point de départ est une définition formelle de la politique de sécurité et de sa violation.

2.2.2.1 Définition des politiques de sécurité

Les auteurs s'intéressent à des politiques d'*intégrité des données*, c'est-à-dire des politiques consistant à spécifier explicitement le(s) utilisateur(s) autorisé(s) à modifier certaines données critiques du système. De telles politiques sont, par exemple :

- Le mot de passe d'un utilisateur ne peut être modifié que par l'utilisateur lui-même ou par l'administrateur ;
- Les journaux système ne peuvent être écrits que par le service de journal système (*syslog*) ;
- Le liste de contrôle d'accès d'un fichier ne peut être modifiée que par le propriétaire du fichier ;
- etc.

L'expression formelle de ces politiques utilise une définition modifiée de la notion de *non-interférence*, introduite initialement par Goguen & Meseguer [22]. La définition originale de Goguen et Meseguer décrit la non-interférence *entre utilisateurs* :

Définition 2.2.1 *Un groupe d'utilisateurs G n'interfère pas avec un groupe G' si la vision de l'état du système de G' est indépendante des actions effectuées par G.*

La définition utilisée par Ko et Redmond est celle de la non-interférence *entre utilisateurs et données* et peut s'énoncer de manière non-formelle ainsi :

Définition 2.2.2 *Un groupe d'utilisateurs G n'interfère pas avec un ensemble de données D si les valeur de D sont indépendantes des actions effectuées par G.*

A titre d'exemple, U étant l'ensemble des utilisateurs, u un utilisateur quelconque, $root$ l'utilisateur administrateur du système et $passwd_u$ le mot de passe de u, la politique « Seuls u et l'administrateur peuvent modifier le mot de passe de u » s'exprime « $U - \{u, root\}$ n'interfère pas avec $passwd_u$. »

Cette définition de la non-interférence est plus restrictive que la définition originale de Goguen & Meseguer : en effet, elle ne permet d'exprimer que l'intégrité, tandis que la définition originale peut être utilisée indifféremment pour exprimer l'intégrité et la confidentialité.

2.2.2.2 Contrôle de la politique

Étant donnée une trace d'exécution du système, la politique de sécurité est par définition respectée si la propriété de non-interférence qui en découle est vérifiée tout au long de la trace.

La méthode de vérification proposée par les auteurs s'appuie sur deux notions. Une opération est dite *privilégiée* si elle est exécutée pour le compte d'un utilisateur autorisé à modifier une donnée sujette à la politique de sécurité (dans l'exemple précédent, les opérations exécutées par u ou $root$ sont privilégiées en ce qui concerne la donnée $passwd_u$). De même, les autres opérations sont *non-privilégiées*[4]. Par extension, les auteurs parlent de processus ou d'utilisateurs privilégiés et non-privilégiés. D'autre part, un état du système, c'est-à-dire une instance des données du système, est dit *sûr*[5] si cet état reflète la politique d'une certaine manière : par exemple, les listes de contrôle d'accès (ACL) dans cet état interdisent aux utilisateurs non-privilégiés de modifier les données soumises à la politique.

Les auteurs montrent que les propriétés de non-interférence se ramènent à trois hypothèses. Nous en présentons ici une interprétation « intuitive » :

1. Si le système se trouve dans un état sûr, aucun utilisateur non-privilégié n'a la possibilité de modifier directement une donnée soumise à la politique. Cela suppose d'une part que la définition du critère de « sûreté » est cohérente avec la politique (en particulier en ce qui concerne les ACL) et d'autre part que l'on fait confiance aux mécanismes de contrôle d'accès.
2. Le système se trouve toujours dans un état sûr. En particulier, si le système se trouve dans un état sûr à l'étape i, alors aucun utilisateur

[4] *unprivileged* dans le texte original en anglais

[5] *safe* en anglais

non-privilégié n'est en mesure d'exécuter une opération menant à un état non-sûr à l'étape $i + 1$, et aucune opération privilégiée présente dans la trace d'exécution ne transforme un état sûr en un état non-sûr. Il en découle qu'en outre, l'état initial du système doit être sûr.

3. Les opérations non-privilégiées peuvent commuter avec les opérations privilégiées, c'est-à-dire étant donné une opération privilégiée op_i et une opération non-privilégiée op_{i+1}, l'exécution de la séquence op_i, op_{i+1} mène à un état du système équivalent à celui produit par l'exécution de la séquence op_{i+1}, op_i.

La troisième règle exprime essentiellement le fait qu'aucun utilisateur non-privilégié ne modifie l'effet des opérations d'un processus privilégié en modifiant, *via* ses propres opérations, l'état dans lequel ces opérations privilégiées seront exécutées. Autrement dit, aucun utilisateur non-privilégie ne détourne l'exécution d'un processus privilégié via une attaque du type *race-conditions*[6].

Les auteurs expriment spécifiquement la condition en termes d'effet « équivalent », et non identique. En effet, l'exigence d'un effet identique serait une contrainte inutilement forte, et par ailleurs difficile à atteindre en pratique. Deux états sont considérés « équivalents » si les valeurs des données sujettes à la politique de sécurité sont égales ; bien que les valeurs d'autres données non-essentielles du point de vue de la politique peuvent différer.

Remarque 2.2.3 *La « commutativité » des opérations n'est donc pas une notion symétrique. Il est requis qu'en exécutant une opération non-privilégiée avant ou après une opération privilégiée, l'effet produit par cette dernière soit « équivalent » ; en revanche, l'effet produit par l'opération non-privilégiée peut être différent dans chaque cas.*

2.2.2.3 Utilisation en détection d'intrusions

Comme les auteurs le soulignent, il a été démontré que dans le cas général, une propriété de non-interférence ne peut pas être contrôlée uniquement par l'analyse de traces d'exécution [23]. Dans le cadre de l'approche proposée, ceci est néanmoins possible, dès lors que l'on connaît la sémantique de chaque opération possible.

Dans l'implémentation réalisée par les auteurs, fonctionnant sous Linux, les opérations sont assimilées aux appels systèmes. Les effets produits par chaque appel système possible étant connus, des conditions de commutativité peuvent être définies pour chaque couple possible d'appels système :

- certains couples d'appels systèmes peuvent commuter inconditionnellement. Par exemple, les auteurs considèrent qu'une occurrence de *close*

[6] Il s'agit d'attaques qui exploitent une fenêtre temporelle permettant d'influer sur l'exécution d'un programme.

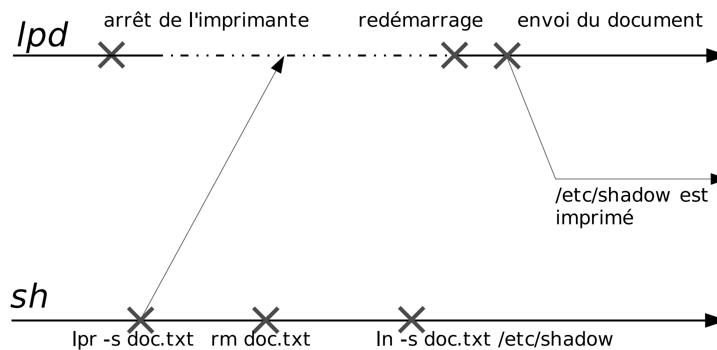


FIG. 2.1 – Attaque contre *lpr*

peut toujours commuter avec l'exécution de tout autre appel, de même qu'une ouverture de fichier en lecture peut toujours commuter avec une autre ouverture en lecture.
- dans les autres cas, la commutativité de deux appels systèmes est soumise à condition. Par exemple, une ouverture de fichier en écriture ne commute avec une autre ouverture en écriture que si leurs fichiers cibles respectifs sont différents.

Les auteurs proposent ainsi de construire une table de commutativité des appels systèmes. L'algorithme de détection consiste alors à vérifier, d'après cette table, que les appels systèmes exécutés par les programmes privilégiés (c'est-à-dire, ceux qui permettent de modifier les données soumises à la politique) commutent avec ceux exécutés par les programmes non privilégiés. En pratique, le détecteur consiste en une sonde qui intercepte les appels système exécutés par les processus applicatifs, et qui alimente l'algorithme de vérification proprement dit.

2.2.2.4 Discussion

Le principal avantage de cette méthode consiste en sa nature, purement paramétrée par la politique. Elle s'appuie sur un modèle du système d'exploitation (la table de commutativité), une spécification formelle de la politique de sécurité (les propriétés de non-interférence) et un théorème permettant de valider une trace d'exécution selon cette politique (sûreté des états, préservation de cette sûreté et commutativité des opérations). En écartant toute donnée empirique, phase d'apprentissage et scénarios d'attaques, elle est en mesure de détecter des violations de la politique par des moyens (*exploits*) connus aussi bien qu'inconnus.

Par ailleurs, l'implémentation se montre relativement légère en comparaison avec d'autres systèmes de détection d'intrusions ; en particulier, la détection a lieu en cours d'exécution, et non *a posteriori*.

Néanmoins, le champ d'application de cette approche est limité à la fois par le spectre des politiques de sécurité pouvant être exprimées (intégrité de *données résidentes*) et la nature des attaques qui violent les critères contrôlés (*race-conditions*). Ce système n'est pas en mesure de détecter des attaques (de type *race-conditions* ou autres) qui violent une politique de *confidentialité*, ou d'intégrité de *données en transit*. Les auteurs citent en exemple l'attaque de type *race-conditions contre lpr*, grâce à laquelle un simple utilisateur (« Alice ») peut contourner l'interdiction d'accéder au contenu de certains fichiers, par exemple */etc/shadow* (cf. figure 2.1) :

1. Alice bloque l'imprimante ;
2. Alice requiert l'impression symbolique d'un fichier qu'elle est autorisée à accéder, par exemple *doc.txt*. Après vérification de ses droits sur *doc.txt*, la requête est acceptée ;
3. l'imprimante étant bloquée, l'impression ne démarre pas ;
4. Alice détruit *doc.txt* et le remplace par un lien symbolique sur */etc/shadow* ;
5. Alice débloque l'imprimante ;
6. le service *lpd* imprime *doc.txt*, c'est-à-dire en réalité */etc/shadow ;*

Cette attaque, du type *race-conditions*, exploite le fait que le contrôle d'accès au document imprimé a lieu à l'étape 2, tandis que l'impression effective a lieu à l'étape 6. Or, le système n'interdit pas de créer un lien symbolique vers un fichier quelconque, même si l'accès à ce dernier n'est pas autorisé, considérant que les droits d'accès au fichier cible s'appliqueront lors de tout accès ultérieur. Donc, entre les étapes 2 et 6, le fichier a pu être remplacé par un lien sur une autre information. Comme il s'agit d'une violation de confidentialité et non d'intégrité, l'approche basée sur la non-interférence entre utilisateurs et données ne peut pas être utilisée pour la détecter.

En conclusion, les possibilités d'utilisation de cette approche se montrent donc assez restreintes, bien que le principe en soit intéressant. Nous souhaitons proposer une approche utilisable de manière plus générale, dont le but serait de détecter les violations d'une politique de sécurité définie par ailleurs sous forme de contrôle d'accès.

2.3 Contrôle d'accès

Le principal moyen d'implémenter une politique de confidentialité et d'intégrité dans un système d'information est le *contrôle d'accès*. Un modèle de contrôle d'accès classique consiste en :

- un ensemble de *sujets*, qui représentent les entités actives du système (par exemple, les processus) ;
- un ensemble d'*objets*, qui représentent des contenants d'information (par exemple, des fichiers) ;

– un ensemble de *privilèges*. Chaque privilège associé au couple (s,x) autorise le sujet s à effectuer une certaine action sur x, x pouvant être soit un objet (par exemple dans le cas des privilèges « lecture » ou « destruction »), soit un autre sujet (par exemple dans le cas du privilège « délégation ») ;
– des *règles de modification* de ces propriétés (création et destruction de sujets, d'objets et de privilèges).

La représentation naturelle d'un tel modèle est sous la forme d'une *matrice de contrôle d'accès*, dont chaque ligne correspond à un sujet, chaque colonne à un objet ou un sujet et chaque élément à un ensemble de privilèges. Le principal avantage de cette vision est son caractère très intuitif. Une exigence de confidentialité interdit la *lecture de l'information* à certains sujets : par exemple, la politique de confidentialité « Le sujet *Alice* n'est pas autorisé à accéder au contenu de l'objet */etc/shadow »* se traduit par l'absence du privilège de lecture dans la case $M[Alice, /etc/shadow]$ de la matrice de contrôle d'accès M. De même, une exigence d'intégrité interdit à certains sujets de *modifier l'information*, ainsi, la politique d'intégrité « Le sujet *Alice* n'est pas autorisé à modifier le contenu de l'objet */etc/password* » s'exprime naturellement par l'absence du privilège d'écriture dans la case $M[Alice, /etc/shadow]$.

Il existe deux grands types de politiques de contrôle d'accès :
– dans le contrôle d'accès *discrétionnaire*, les propriétés concernant chaque information sont entièrement définies par son propriétaire ;
– dans le contrôle d'accès *mandataire*[7], des propriétés plus fortes (et, généralement, restrictives) s'appliquent et ne peuvent être modifiés que par certains sujets privilégiés.

L'objectif principal du contrôle d'accès mandataire consiste à imposer des restrictions aux propriétaires des informations quant au transfert de leurs privilèges. Dans la suite de cette section, nous présentons le modèle général dit « HRU » et les problèmes qui lui sont intrinsèquement liés, puis nous discutons de modèles mandataires visant à prévenir ces problèmes.

2.3.1 Modèle général de type *HRU*

Les politiques de contrôle d'accès traditionnelles sont basées sur le modèle défini par Harrison, Ruzzo et Ullman dans [24], dit « *HRU* ». Dans ce modèle, les sujets sont eux-mêmes des objets, c'est-à-dire, chaque sujet possède à la fois des droits d'accéder à certains objets, et des droits de modifier la matrice de contrôle d'accès en créant ou détruisant des sujets, des privilèges de certains sujets sur certains objets. Étant donnée une matrice de contrôle

[7]Nous utilisons ce terme car il fait désormais partie de l'usage courant. C'est en réalité une traduction inexacte du terme anglais « m*andatory access control* », qui signifie plus précisément « contrôle d'accès *obligatoire* ».

d'accès M, une trace d'exécution est une séquence d'opérations, dont chacune est de la forme :

si $a_1 \in M[s_1, x_1]$, $a_2 \in M[s_2, x_2]$, ..., $a_n \in M[s_n, x_n]$

alors s_1 effectue a_1 sur x_1, ..., s_n effectue a_n sur x_n.

Du point de vue de la politique de contrôle d'accès, une exécution est légale si pour chaque action, le sujet concerné possède le privilège requis sur l'objet ou le sujet visé.

Or, bien qu'une exécution soit légale, elle peut néanmoins aboutir à une violation de la politique de sécurité. On dit alors qu'un système est sûr si aucune des exécutions permises par les règles de la politique de contrôle d'accès ne peut conduire à une violation des propriétés de celle-ci. De façon générale, les politiques de contrôle d'accès présentent deux classes de problèmes : le *transfert des privilèges* (« access leak ») [25] et la *divulgation d'information* (« information leak »).

2.3.1.1 Transfert de privilèges

Une politique de contrôle d'accès dans laquelle un sujet s **ne possède pas** le privilège d'exécuter une action a[8] sur un objet o est par définition violée si s acquiert ce privilège.

Considérons par exemple la politique suivante :

	o	s_a	s_b
s_a	$\emptyset$	*	$\emptyset$
s_b	lecture	délégation	*

Contrairement à s_b, s_a n'est pas autorisé à lire o. Toutefois, s_b est autorisé à déléguer des privilèges à s_a. Ainsi, l'exécution de l'unique opération :

si *delegation* $\in M[s_b, s_a]$

alors s_b crée *lecture* dans $M[s_a, o]$

est légale et conduit à la matrice :

	o	s_a	s_b
s_a	lecture	*	$\emptyset$
s_b	lecture	délégation	*

Ainsi, le privilège lecture sur o a été transféré à s_a, ce qui constitue une violation de la politique initialement spécifiée. Le système n'est donc pas sûr.

Dans [24], les auteurs présentent un algorithme de vérification de sûreté pour un système *mono-opérationnel*, c'est-à-dire un système où chaque opération consiste en une unique action (c'est le cas dans l'exemple ci-dessus).

[8] Dans la suite du texte, nous le désignerons simplement par « le privilège a ».

L'algorithme consiste à tester un ensemble fini d'exécutions ; un théorème simple garantit que si aucune de ces exécutions n'aboutit à un transfert de privilège violant la politique initiale, alors le système est sûr.

Dans le cas pratique, cependant, ce test de sûreté reste difficilement utilisable. D'une part, la complexité de cet algorithme est de classe *NP-complet*. D'autre part, l'hypothèse d'un système mono-opérationnel ne correspond pas aux systèmes réels. Or, les auteurs démontrent que si le système n'est *pas* mono-opérationnel, alors le test de sûreté devient un problème *indécidable*, car équivalent à la prédiction de l'arrêt d'une machine de Turing. Or, un système mono-opérationnel n'est pas utilisable en pratique. Dans le cas général, une politique de sécurité exprimée sous forme de modèle *HRU* peut donc potentiellement donner lieu à un transfert illégal de privilèges qu'il demeure impossible de prévenir.

2.3.1.2 Divulgation d'information

Une politique de contrôle d'accès dans laquelle un sujet s **ne possède pas** le privilège d'accéder à l'information contenue dans un objet o peut également être violée si s possède le privilège d'accéder à l'information contenue dans un objet o' et si le contenu de o est transféré dans o'. Un exemple pratique est l'attaque contre *lpr*, que nous avons présentée précédemment (cf. figure 2.1) :

- Alice peut légalement demander l'impression de *doc.txt* ;
- Alice peut légalement supprimer *doc.txt* ;
- Alice peut légalement créer un lien sur */etc/shadow* appelée *doc.txt* ;
- le service *lpd* peut légalement lire */etc/shadow* et envoyer son contenu à l'imprimante.

Formellement, dans la politique suivante :

	o_m	o_n
s_a	$\emptyset$	lecture
s_b	lecture	écriture

le sujet s_a n'est pas autorisé à accéder au contenu de o_m. Cependant, la politique autorise l'exécution consécutive des trois opérations :

1. **si** $lecture \in M[s_b, o_m]$
   alors s_b lit o_m ;
2. **si** $ecriture \in M[s_b, o_n]$
   alors s_b écrit o_n ;
3. **si** $lecture \in M[s_a, o_n]$
   alors s_a lit o_n.

Or, à la suite de l'opération 2, le contenu de o_n reflète celui de o_m. Par conséquent, l'exécution de l'opération 3 est équivalente à la lecture de o_n par s_a, ce qui est contraire à la politique. Le système n'est donc pas sûr.

La définition de la notion même de divulgation d'information s'avère difficile. En effet, l'information divulguée peut consister en l'intégralité du contenu de o, ou d'une partie de celui-ci (éventuellement un unique bit), ou encore une autre information déduite de l'information originale (par exemple sa valeur de hachage).

Si un sujet s n'est pas autorisé à accéder au contenu d'un objet o, ce dernier peut donc néanmoins lui être potentiellement divulgué dès lors qu'il existe un sujet s' autorisé à accéder au contenu de o, et un objet o' accessible par s et modifiable par s'. Or, la possibilité de l'obtention de tels privilèges par s et s' est indécidable dans le cas général, la possibilité d'une divulgation du contenu de o à s est donc également indécidable.

2.3.1.3 Garanties de sécurité

Le modèle *HRU* est le modèle de contrôle d'accès le plus général, mais l'impossibilité de vérifier la sûreté d'un système dans le cas général est un inconvénient majeur. Plusieurs travaux ont visé à remédier à ce problème.

Sandhu a proposé dans [26] d'introduire la notion de *type* : chaque objet appartient à un certain type, de même qu'il existe des types de sujets. Lorsqu'un objet ou un sujet est créé, son type est défini et n'est jamais modifié. Les opérations autorisées sur la matrice de contrôle d'accès dépendent alors des types de sujets et objets concernés. Sandhu a montré que la sûreté d'un système à matrice de contrôle d'accès typée est décidable si le système est *monotone*, c'est-à-dire s'il est impossible de supprimer définitivement des privilèges.

Dacier et Deswarte ont étendu ce modèle en proposant de considérer le *graphe de privilèges* [27]. Dans ce graphe, les noeuds représentent les états de sûreté du système, c'est-à-dire les états de la matrice de contrôle d'accès. Les arcs du graphe représentent les transitions d'états par transferts de privilèges. Ces transferts peuvent être conformes à la politique de sécurité en utilisant les opérations de modification de la matrice, ou illégaux, basés sur des attaques exploitant des vulnérabilités connues du système. En pondérant chaque arc par une estimation de la complexité et du coût de réalisation de la transition correspondante (légale ou par l'intermédiaire d'une attaque), il est possible de vérifier la sûreté du système et d'évaluer quantitativement les éventuels risques de violation [28].

2.3.2 Contrôle d'accès mandataire

Le but du contrôle d'accès mandataire consiste à prévenir les risques de violations par transfert de privilèges et/ou par divulgation d'informations précisément en interdisant ou limitant la libre création/suppression de privilèges. Cela revient à imposer une politique prédéfinie visant à garantir la sûreté du système au prix d'une perte de la flexibilité et de l'universalité.

2.3.2.1 Modèle « Bell & LaPadula »

Le modèle proposé par Bell et LaPadula [29], dit « BLP », permet de prévenir les divulgations d'informations. Il repose sur le modèle *HRU* et exclut toute création ou destruction de sujets ou d'objets.

Une politique basée sur le modèle *BLP* définit n niveaux de sécurité. S étant l'ensemble de tous les sujets et O celui de tous les objets, une fonction $F : S \cup O \rightarrow \{1 \dots n\}$ associe un niveau de sécurité à chaque sujet (on parle alors de *« niveau d'habilitation*[9] ») et à chaque objet (*« niveau de classification*[10] »).

La matrice de contrôle d'accès M doit vérifier les deux règles suivantes :

1. Si $lecture \in M[s, o]$ alors $F(s) \geq F(o)$;
2. Si $lecture \in M[s, o_1]$ et $ecriture \in M[s, o_2]$ alors $F(o_1) \geq F(o_2)$.

Informellement, ce modèle attribue à chaque sujet un « pouvoir d'accès » sous forme de niveau d'habilitation et une « restriction d'accès » à chaque information contenue dans un objet sous forme de niveau de classification. La première règle (« *propriété simple* ») assure la confidentialité de l'information en exigeant qu'un sujet ne puisse accéder à un objet que si son « pouvoir d'accès » est suffisant, c'est-à-dire si son habilitation est au moins égale à la « classification » de l'objet. D'après la seconde règle (« *propriété* * »), 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 : en effet, une divulgation d'information (c'est-à-dire, sa lecture ou la lecture d'une copie de celle-ci, par un sujet non autorisé initialement) nécessiterait de la rendre accessible sous une classification *inférieure* à sa classification originale.

Afin de vérifier la propriété *, il est nécessaire de pouvoir contrôler tous les flux d'informations entre objets que les sujets peuvent créer. Ceci pose d'importants problèmes pratiques. L'existence éventuelle de canaux « cachés » peut rendre ce contrôle impossible. Considérons par exemple la séquence suivante :

1. s lit depuis o_1 ;
2. s lit depuis o_3 ;
3. s écrit dans o_2.

Si $F(o_1) < F(o_2) < F(o_3)$, la politique sera respectée si à l'étape 3, s écrit dans o_2 l'information lue dans o_1. En revanche, la politique sera violée si s écrit dans o_2 l'information lue dans o_3. Or, si l'information est stockée par exemple dans un tampon en mémoire où les modifications et transferts ne peuvent pas être contrôlés, il n'est pas possible de déterminer sa provenance.

---

[9] *clearance level* en anglais
[10] *classification level* en anglais

Afin de prévenir ce problème, une version plus restrictive de la politique *BLP* utilise les règles suivantes :

1. Si *lecture* $\in M[s, o]$ alors $F(s) \geq F(o)$;
2. Si *ecriture* $\in M[s, o]$ alors $F(s) \leq F(o)$.

Dans cette version de la politique, la séquence de l'exemple précédent est strictement illégale : en effet, si s peut lire o_3, alors $F(s) \geq F(o_3)$, donc $F(s) > F(o_2)$, donc s n'est pas autorisé à écrire dans o_2.

On peut remarquer, toutefois, que le modèle est rarement appliqué tel quel. D'une part, il n'autorise que des communication à sens unique, des sujets à faible habilitation vers les sujets à habilitation élevée. Si, en pratique, un retour d'informations s'avère nécessaire, on doit introduire des exceptions à la politique, en remettant potentiellement en cause ses garanties de sûreté. D'autre part, certains sujets doivent être autorisés à modifier les niveaux de habilitation et de classification, c'est-à-dire modifier la fonction M. Dans ce cas, le transfert de privilèges ou la divulgation d'informations ne sont plus exclus. La prévention de ce problème exige de modéliser la *sécurité des transitions*, en définissant un ensemble (éventuellement vide) de sujets et d'objets dont l'habilitation ou la classification peut être modifiée [30].

2.3.2.2 Modèle « Biba »

De modèle dit « Biba » [31] vise à garantir *l'intégrité*. Il s'agit en réalité d'un modèle similaire à *BLP*. Chaque sujet dispose à nouveau d'un « pouvoir d'accès » qui correspond ici à son « niveau d'intégrité », et la modification de chaque objet requiert un certain niveau d'intégrité. Les règles relatives à la matrice de contrôle d'accès n'autorisent la modification du contenu d'un objet qu'aux sujets possédant un niveau d'intégrité suffisant. D'autre part, la communication *entre sujets* est prise spécifiquement en compte : on entend par « *invocation de s_2 par s_1* » un flux d'informations unidirectionnel depuis s_1 vers s_2.

1. Si *lecture* $\in M[s, o]$ alors $F(s) \leq F(o)$;
2. Si *ecriture* $\in M[s, o]$ alors $F(s) \geq F(o)$;
3. Si *invocation* $\in M[s_1, s_2]$ alors $F(s_1) \geq F(s_2)$.

Si tous les canaux de flux d'informations apparaissent sous forme d'objets, alors une invocation de s_2 par s_1 s'apparente à une écriture par s_1 dans un certain objet o suivie d'une lecture de o par s_2. Dans ce cas, la troisième règle découle directement des deux précédentes.

Les mêmes remarques que celles concernant *BLP* s'appliquent ici.

2.3.2.3 Muraille de Chine

Le modèle dit « Muraille de Chine » proposé par Brewer et North dans [32] où l'ensemble des objets est divisé en *classes de conflit d'intérêt*, chaque

classe étant par ailleurs divisée à son tour en *ensembles de données*. Un sujet n'est autorisé à accéder à deux objets différents que si ceux-ci appartiennent au même ensemble de données, ou s'ils appartiennent à deux classes de conflit d'intérêts différentes.

Ce modèle vise à garantir la confidentialité en interdisant des divulgations d'informations au sein de chaque classe de conflit d'intérêts. Il correspond donc à des besoins commerciaux spécifiques telles que ceux des applications bancaires ou des cabinets de conseil. Sandhu a montré qu'il était possible d'étendre le modèle en adoptant une politique multi-niveaux de type *BLP* [33].

2.3.3 Le modèle « Take-Grant »

Le modèle « Take-Grant » est un modèle de protection général résultant de travaux menées depuis les années 1970 [34]. Son objectif est de vérifier la sûreté des politiques de confidentialité et d'intégrité, en tenant compte à la fois du contrôle d'accès, des possibilités de transfert de privilèges et des flux d'informations.

Ce modèle fournit une base théorique adaptée en particulier à l'étude des politiques de sécurité d'un point de vue des risques de vol de privilèges, de divulgation d'information et de conspiration entre sujets [35].

2.3.3.1 Modélisation des politiques de sécurité

Une politique de confidentialité/intégrité dans le modèle « Take-Grant » est modélisée par un graphe orienté appelé *graphe de protection* où :
- chaque noeud représente un sujet ou un objet ;
- un arc reliant un noeud x à un noeud y représente les droits de x sur y.

Toute modification des règles de contrôle d'accès, c'est-à-dire toute création, destruction ou transfert de droits, se traduit par des opérations de transformation du graphe de protection. Le modèle définit quatre primitives de transformation :

create : « Le sujet x crée y » ou « Le sujet x crée des droits sur y ». Si y n'existe pas, le graphe résultant possède un nouveau noeud y et un nouvel arc reliant x à y, sinon, l'arc reliant x à y est remplacé dans le graphe résultant par un nouvel arc de x vers y correspondant au nouvel ensemble de droits de x sur y ;

remove : « Le sujet x supprime des droits sur y ». Cette opération est symétrique de la précédente ;

take : « Le sujet x obtient de y les droits α sur z ». Si y possède les droits α sur z (c'est-à-dire s'il existe un arc de y vers z correspondant à un ensemble de droits incluant α) et si x possède le droit « *take* » sur y

(c'est-à-dire, si x est autorisé à s'approprier des droits associés à y), le graphe résultant possède un arc de x vers z correspondant à l'ensemble de droits α ;

grant : « Le sujet x accorde à y les droits α sur z ». Si x possède les droits α sur z (il existe un arc de x vers z correspondant à un ensemble de droits incluant α) et si x possède le droit g « *grant* » sur y (c'est-à-dire, si x est autorisé à transmettre ses droits à y), le graphe résultant possède un arc de y vers z correspondant à l'ensemble de droits α.

Ces opérations représentent des modifications du contrôle d'accès *explicitement autorisées au regard de la politique de sécurité* : en effet, l'application de l'opération *grant*, par exemple, suppose que les règles de la politique autorisent x à introduire la modification consistant à donner un certain droit à y. Par conséquent, tout graphe de protection construit à partir d'un graphe initial par l'application de ces règles est conforme aux règles de la politique de sécurité.

Pour cette raison, ces règles sont appelées des *transformations « de jure »*.

2.3.3.2 Divulgation d'informations

Le modèle original, définissant le graphe de protection et les quatre opérations « *de jure* », a été étendu par Bishop et Snyder [36] afin de prendre en compte les divulgations d'informations possibles selon une politique de sécurité. En effet, si la politique interdit à un sujet x de lire y, x peut dans certains cas accéder par exemple à une copie de y et, de fait, lire le *contenu* de y.

Le principe consiste à interpréter les arcs correspondant à des ensembles de droits incluant la lecture (*read*) et/ou l'écriture (*write*) en tant que canaux de flux d'informations. Bishop et Snyder identifient quatre types de transfert d'informations potentiels et définissent les transformations correspondantes du graphe de protection. Ces transformations sont appelées *transformations « de facto »* :

post : « z envoie des informations à x à travers y ». Si z peut écrire dans y et x peut lire y, alors il existe dans le graphe résultant un arc de x vers z portant le droit de *lecture* (« x peut lire z ») ;

pass : « y transmet depuis z dans x ». Si y peut lire z et écrire dans x, alors le graphe résultant possède un arc de x vers z portant le droit de *lecture* ;

spy : « x écoute z à travers y ». Si y peut lire z et x peut lire y, alors le graphe résultant possède un arc de x vers z portant le droit de *lecture* ;

find : « x retrouve z par l'intermédiaire de y ». Si z peut écrire dans y et y dans x, alors le graphe résultant possède un arc de x vers z portant le droit de *lecture*.

Ces quatre transformations correspondent à des *interprétations* des règles de contrôle d'accès et non à des modifications de celles-ci. Pour cette raison, les arcs créés par les opérations *de facto* sont qualifiés d'*implicites*, par opposition aux arcs *explicites* construits par les opérations *de jure*.

2.3.3.3 Validation des politiques

Le modèle « Take-Grant » a pour objectif l'analyse des politiques de sécurité. Des possibilités de transfert ou de vol de privilèges dans une politique de sécurité peuvent être aisément détectées. En effet, une politique rend par définition possible l'obtention par x d'un droit α sur y s'il est possible de construire, à partir de son graphe de protection, un graphe dans lequel il existe un arc de x vers y représentant le droit α, uniquement en appliquant les transformations *de jure*.

Les auteurs montrent par deux théorèmes que la preuve de l'existence ou non d'une telle série de transformations se réduit à la recherche de chemins dans le graphe de protection, composés d'arcs correspondant à des droits *take* ou *grant*. Intuitivement, cela signifie que si s possède un droit α sur y et si la politique permet un enchaînement de délégations ou d'appropriations, permettant soit à s de transmettre ce droit à x, soit à x de se l'approprier, alors x est en mesure d'obtenir le droit α sur y.

De même, le risque de divulgation d'une information équivaut à l'existence d'une série de transformations *de jure* et *de facto* du graphe de protection, aboutissant à la création de l'arc (éventuellement implicite) correspondant. Une variation des théorèmes cités ci-dessus s'applique ; il suffit donc là encore de rechercher certains chemins dans le graphe pour détecter ces risques.

2.3.3.4 Graphe d'action et conspiration

Bishop et Snyder proposent par ailleurs une méthode d'analyse des possibilités de *conspiration* [35] : elle permet de déterminer le nombre minimum de sujets participants, nécessaire pour rendre possible une divulgation d'information.

Pour cela, les auteurs définissent un *graphe d'action*[11], qui modélise les possibles flux d'informations *entre sujets*. Par construction, ce graphe orienté est tel que :

- chaque noeud correspond à un sujet ;
- il existe un arc d'un noeud u vers un noeud u' si u peut obtenir de l'information depuis u' directement ou au moyen des transformations *de facto*.

Conformément au résultat intuitif, les auteurs démontrent que le nombre minimum nécessaire de sujets coopérants pour divulguer l'information y à

[11] *Acting graph* en anglais.

un sujet u correspond au chemin le plus court dans le graphe d'action entre le noeud u et le noeud correspondant à un sujet pouvant lire y.

2.3.3.5 Discussion

Le modèle « Take-Grant » fournit une base théorique riche pour la modélisation des politiques de confidentialité/intégrité et la vérification de leurs propriétés. Il est possible de l'utiliser pour détecter des anomalies ou des erreurs dans la spécification de la politique, notamment le risque de divulgation d'une information ou la possibilité d'obtention d'un droit par un utilisateur. La principale application de ce modèle consiste donc à *concevoir une politique de sécurité*.

Cependant, des politiques « non-sûres » du point de vue du modèle « Take-Grant » restent largement présentes en pratique. Par exemple, tout système de type Unix implémente la politique suivante :

fichier	propriétaire	droits du propriétaire	droits des autres
/etc/shadow	root	lecture,écriture	$\emptyset$
/etc/resolv.conf	root	lecture,écriture	lecture

Le modèle « Take-Grant » indique qu'un simple utilisateur est en mesure de prendre connaissance du contenu de */etc/shadow* : il suffit pour cela que *root* le copie dans */etc/resolv.conf*. Or, cela constitue une violation de la politique de sécurité.

2.4 Contrôle des flux d'informations

Le contrôle de flux d'informations offre le cadre le plus général permettant de construire une politique de confidentialité et d'intégrité. Il existe différentes approches de cette problématique.

2.4.1 Treillis de Denning

Les premiers travaux dans ce domaine visaient à contrôler et/ou prévenir la totalité des flux d'informations pouvant avoir lieu entre différents conteneurs d'informations dans un système générique. Un modèle a été proposé par Denning dans [37]. Ce modèle permet de représenter en particulier des politiques de sécurité du type *BLP*, *Biba* et *Muraille de Chine*.

2.4.1.1 Représentation du système

Dans ce modèle, un système est assimilé à un treillis $(N, P, SC, \oplus, \rightarrow)$, où :

– N est un ensemble de *conteneurs d'information* (fichiers, variables, registres, objets, utilisateurs etc...) ;
– P est un ensemble de *processus* qui créent les flux d'informations ;
– SC est un ensemble de *classes de sécurité*. Chaque conteneur et chaque processus appartient à exactement une classe de sécurité. Pour les utilisateurs et les processus, les classes de sécurité correspondent à des niveaux de habilitation, pour les autres objets, elles *peuvent* représenter des niveaux de classification ;
– $\oplus$ est un opérateur binaire, définissant la classe de sécurité à laquelle appartient le résultat d'une fonction binaire, selon les classes de ses opérandes. Si une fonction est appliquée à deux paramètres a et b, la classe de son résultat est $classe(a) \oplus classe(b)$;
– $\rightarrow$ est un prédicat binaire, qui définit les flux d'informations autorisés. Étant donné deux classes A et B, $A \rightarrow B$ signifie que les flux des conteneurs de classe A vers les conteneurs de classe B sont autorisés.

L'association des conteneurs aux classes de sécurité peut être soit statique, c'est-à-dire définie par la politique et invariable, soit dynamique en fonction du contenu des conteneurs. Le modèle de contrôle d'accès mandataire de Bell & LaPadula [29], que nous avons décrit dans la section 2.3.2.1, est un exemple de contrôle de flux d'informations à association statique.

2.4.1.2 Contrôle des flux

Dans le cas d'un modèle à association statique, il est possible en principe de contrôler les flux d'informations à l'exécution, ou d'effectuer une certification statique.

Dans le premier cas, il est nécessaire de pouvoir observer tout transfert d'informations au niveau le plus élémentaire [38], ou d'assurer des tests exhaustifs de tous les programmes concernés. Un tel contrôle n'est donc pas possible en pratique dans un environnement général traditionnel (Linux, Unix, Windows), notamment dans le cas de programmes disponibles uniquement sous forme binaire. Des modèles abstraits d'architecture permettant un tel contrôle ont été proposés [39].

La certification statique vise à garantir qu'un programme donné ne peut pas produire un flux d'informations illégal, en utilisant les connaissances disponibles sur ce programme. Malheureusement, ces connaissances sont généralement insuffisantes dans le cas de programmes écrits dans des langages classiques comme le C, *a fortiori* dans le cas de programmes disponibles uniquement sous forme binaire. Par exemple, les flux d'informations produits par un débordement de tampon (*buffer overflow*) ou un débordement d'index (*out-of-array*) dépendent du type de processeur et de compilateur utilisé, mais aussi de l'état précis du système à l'instant où le flux a lieu. Une analyse statique ne peut donc pas les mettre en évidence.

De nombreux travaux ont porté sur ce dernier point, dans le but de

développer des langages de programmation « sûrs » du point de vue des flux d'informations, grâce à des systèmes de typage [40], une architecture excluant tout effet de bord [41], etc. Cependant, ces approches ne permettent pas de résoudre le problème dans le cas de programmes existants, implémentés dans des langages non spécifiquement conçus dans ce but.

2.4.2 Non-interférence

La non-interférence [42, 22] est un modèle général de politique de sécurité, dont le but est de vérifier l'*absence de flux d'informations*. Cette notion généralise les notions de confidentialité et d'intégrité ; aussi, les modèles *BLP* et *Biba* (entre autres) peuvent être reformulés sous forme de propriétés de non-interférence.

Un système est représenté ici sous forme d'un tuple $(S, \Sigma, O, Z, C, T, P)$:

- S est un ensemble de sujets ;
- Σ est un ensemble d'états ;
- O est un ensemble de sorties ;
- Z est un ensemble de commandes ;
- $C = S \times Z$ est l'ensemble des opérations de changement d'état ;
- $T : C \times \Sigma \to \Sigma$ est une fonction de transition d'état ;
- $P : C \times \Sigma \to O$ est la fonction définissant les sorties du système.

La propriété de non-interférence est alors définie de la manière suivante :

Définition 2.4.1 *Étant donné deux groupes distincts de sujets G et G' et un ensemble de commandes $A \subseteq Z$, le groupe G n'interfère pas avec le groupe G' en exécutant les commandes de l'ensemble A si et seulement si quels que soient la séquence de commandes $c_s \subseteq C*$ et le sujet $s' \in G'$,*

$$proj(s', c_s, \sigma_i) = proj[s', \pi_{G,A}(c_s), \sigma_i]$$

où

- *$\sigma_i \in \Sigma$ est le i-ème état atteint par le système ;*
- *$proj(s', c_s, \sigma_i) \in O*$ ("projection") est le sous-ensemble ordonné des sorties que le sujet s' est autorisé à lire, en préservant l'ordre d'apparition de ces sorties au cours des transitions d'état (c_s, σ_i) ;*
- *$\pi_{G,A}(c_s)$ est la sous-séquence de commandes obtenues en supprimant de c_s toutes les commandes (s, z) telles que $s \in G$ et $z \in A$.*

Intuitivement, cette propriété signifie que le groupe G n'interfère pas avec G' en utilisant les commandes de A si G' observe toujours une séquence des sorties égale à celle qui aurait lieu si G n'avait exécuté aucune des commandes de A. En d'autres termes, G exécutant les commandes de A n'interfère pas avec G' si et seulement si ces actions de G n'ont aucune influence sur ce que peut observer G'.

La non-interférence est une propriété à la fois très générale et très forte. En utilisant le théorème dit du « déroulement »[12], il est possible de vérifier si un système est *a priori* sûr vis-à-vis d'une propriété de non-interférence. Malheureusement, la non-interférence n'est pas utilisable dans notre cas, elle ne peut pas être contrôlée par une analyse de traces : en effet, il est impossible de vérifier la non-interférence sur une séquence donnée, il est nécessaire de prendre en compte *toutes* les séquences possibles [23].

Le modèle de contrôle de flux d'informations général proposé par McLean [43] étend et généralise cette propriété, ainsi que d'autres modèles de politiques s'y rattachant (non-déductibilité, non-interférence généralisée aux systèmes non-déterministes, séparabilité, etc...)

2.4.3 Modèles pour environnements spécifiques

Les modèles de contrôle de flux d'informations ont trouvé de nombreuses applications dans des environnements particuliers. Le point commun à ces modèles est le fait qu'un flux d'informations n'est possible que s'il emprunte un canal connu, par exemple :

- des environnements orientés objets où l'ensemble des canaux légaux découle des règles de visibilité des méthodes et attributs [44, 45] ;
- des environnements orientés objets, où des flux d'informations sont autorisés par des mécanismes de transfert explicites, par exemple des « privilèges de flux » (*waivers*) associées à des méthodes ou des objets [46] ;
- des environnements de contrôle d'exécution de code mobile, visant principalement à isoler un programme « étranger » du reste du système (*sandbox*) [47] ;
- etc.

Ces modèles sont utilisés essentiellement par des architectures logicielles destinées à garantir la sécurité d'applications commerciales. Cependant, ces modèles sont également basés sur des propriétés spécifiques des environnements d'exécution utilisés, qui permettent notamment d'observer la totalité des flux possibles (par exemple, l'appel de méthodes en *Java*). Un tel modèle serait donc applicable à un système de type Linux, Unix ou Windows s'il était possible d'identifier la totalité des flux d'informations. Ceci est en pratique impossible en considérant des flux à l'échelle des variables et registres élémentaires, mais l'est à l'échelle des objets et fichiers. Nous proposons d'adopter une approche de ce type dans le chapitre 3.

2.4.4 Confinement de processus (*DTE*)

Le modèle « Domain and Type Enforcement » (*DTE*) est un modèle de contrôle d'accès de haut niveau [48], qui synthétise un grand nombre d'élé-

[12] *unwinding theorem* en anglais

ments des modèles cités précédemment. Toutefois, le principal objectif de DTE n'est pas de garantir la sûreté d'une politique de contrôle d'accès, mais de prévenir les abus de privilèges et assure le *confinement des programmes privilégiés* afin d'empêcher leur détournement par des attaquants exploitant des « *rootkits*[13] ». Il constitue une plate-forme sur laquelle peuvent être implémentées des politiques de contrôle d'accès avancées de différents types, en particulier mandataire (*MAC*), dont Bell & LaPadula et Biba.

Le système consiste en un modèle de contrôle d'accès générique (*DTE* proprement dit) et un langage de spécification des politiques (*DTE Language* ou *DTEL*).

2.4.4.1 Définition des politiques de sécurité

Dans un système Linux classique, les programmes privilégiés sont ceux dont l'exécutable appartient à *root* et dont le bit *setuid* est armé. Ces programmes sont utilisés pour limiter les opérations possibles d'un sujet sur un objet (par exemple, il ne doit être possible d'écrire dans le fichier */etc/shadow* que dans le but de modifier *son propre* mot de passe). Si l'attaquant parvient à détourner un de ces programmes (par exemple, grâce à un *buffer-overflow*), il est en mesure d'exécuter n'importe quelle opération ou d'obtenir une session « *root* ».

Afin de prévenir ce type d'attaque, les politiques implémentées par DTE consistent donc à :

1. restreindre les possibilités d'action des programmes privilégiés en leur appliquant le « principe du moindre privilège » ;
2. interdire l'accès aux données « sensibles » autrement que par le moyen des programmes prévus et contrôlés.

Il est par exemple possible d'utiliser DTE pour confiner le programme privilégié */bin/passwd*, grâce auquel les utilisateurs peuvent changer de mot de passe. Dans ce cas, la politique implémentée limite les droits de ce programme à la lecture des bibliothèques système nécessaires à son fonctionnement (*/lib/libc.so,* etc.) et la lecture/écriture du fichier des mots de passe */etc/shadow*. Simultanément, ce dernier fichier est accessible en lecture par les services d'authentification.

2.4.4.2 Modèle de contrôle d'accès

DTE utilise une approche à base de matrice de contrôle d'accès, c'est-à-dire du type « sujet/objet ». Les notions classiques de « sujet » et « objet », telles qu'elles sont présentes par exemple dans le mécanisme discrétionnaire standard, sont ici remplacées respectivement par les *domaines* et les *types*.

[13] On appelle *rootkit* un outil logiciel grâce auquel l'utilisateur est en mesure d'obtenir les privilèges administrateur (« *root* ») sans authentification.

Chaque objet du système[14] possède un type. Chaque processus agissant sur les objets s'exécute dans un domaine, dont découlent ses droits d'accès aux objets.

Le système DTE fait ainsi appel à trois tables :

1. la **table de typage**, qui assigne un type à chaque objet ;
2. la **table de définition des domaines (DDT)** constitue la matrice d'accès proprement dite. Elle détermine les droits (*lecture*, *écriture*, *exécution*, *ajout*, *suppression*) sur les objets de chaque type dans chaque domaine ;
3. la **table d'interaction des domaines (DIT)** définit, pour chaque domaine, les opérations autorisées sur les processus s'exécutant dans les autres domaines (*création*, *destruction*, *envoi de signal*).

Chaque domaine possède un ensemble de « points d'entrée » : en pratique, ce sont les programmes exécutables. Lorsqu'un processus invoque l'appel système *execve* (ou une de ses variantes) pour appeler un fichier exécutabe qui est un point d'entrée de domaine, le processus peut changer de domaine. Dans l'implémentation actuelle de DTE, modes sont possibles :

automatique (auto) : l'exécution du processus se poursuit automatiquement dans le domaine associé à l'exécutable ;

sur demande (exec) : le processus appelant sélectionne le domaine dans lequel son exécution se poursuivra. Ceci se fait à l'aide d'une version particulière de l'appel système *execve* acceptant ce paramètre supplémentaire ;

aucun changement (null) : l'exécution du processus se poursuit automatiquement dans le domaine courant.

Pour chaque exécutable, le mode utilisé peut être défini en fonction du domaine d'exécution du processus appelant.

La politique exprimée ci-dessus, contrôlant l'accès au fichier */etc/shadow*, peut par exemple être implémentée ainsi :

Typage :

- */etc/shadow* est de type *password_t* ;

DDT :

- le type *password_t* est accessible en lecture/écriture dans un domaine *changepasswd_d*, et en lecture dans un domaine *login_d* ;
- l'exécutable */bin/passwd* est un point d'entrée de *changepasswd_d* avec changement automatique de domaine ;
- l'exécutable */bin/login* est un point d'entrée de *login_d* avec changement automatique ;

[14]On entend ici par « objet » toute entité nommée dans le système d'exploitation : fichier, message, mémoire partagée, socket...

DIT :

- A partir de tous les domaines utilisateur, la création de processus dans *changepasswd_d* est autorisée ;
- Depuis les domaines utilisateur, la destruction de processus dans *changepasswd_d* ou l'émission de signaux à leur intention est interdite ;
- Aucune interaction entre domaines n'est autorisée depuis *changepasswd_d.*

La rédaction du fichier de configuration en *DTEL* correspondant incombe à l'administrateur du système et les utilisateurs ne peuvent y déroger. Ceci fait de DTE un modèle mandataire.

2.4.4.3 Contrôle de l'exécution

Si le typage et la DDT assurent le contrôle d'accès à proprement parler, les points d'entrée et les interactions entre domaines définies dans la DIT autorisent une forme de prévention des violations de la politique à base de spécification de comportement (voir section 2.2.1). En effet, chaque domaine ne permet que l'exécution d'une gamme limitée d'opérations. D'autre part, l'ensemble des points d'entrée dont l'accès est autorisé dans ce domaine restreint de la même manière la gamme des *suites d'exécution* possibles.

Par exemple, si l'on définit n domaines $d_1 \ldots d_n$ tels que :

- o_i soit l'ensemble des opérations autorisées dans le domaine d_i (c'est-à-dire, l'ensemble des accès aux objets autorisés en fonction de leurs types et des droits sur ceux-ci associés au domaine d_i) ;
- chaque domaine d_i possède un unique point d'entrée e_i, qui est le seul dont l'invocation soit autorisée dans le domaine d_{i-1},

alors, le démarrage d'un processus dans le domaine d_1 revient de fait à forcer l'exécution de zéro ou plusieurs opérations de o_1, suivie éventuellement de l'exécution de e_2 puis de zéro ou plusieurs opérations o_2, et ainsi de suite. Ceci correspond à la spécification :

$$o_1^*(stop|exec\, e_2 o_2^* \ldots (stop|exec\, e_i o_i^* \ldots (stop|exec\, e_n o_n^*))).$$

Ce schéma de spécification élémentaire peut être enrichi ou raffiné de différentes manières. S'il est autorisé, dans un domaine d_i, d'invoquer les points d'entrée de plusieurs autres domaines $d_{i+1}, d'_{i+1}...$, il en résulte autant de branches parallèles dans la spécification. Inversement, déclarer un exécutable comme point d'entrée d'un nouveau domaine permet de contraindre ses exécutions possibles en affectant à ce domaine les plus petits privilèges nécessaires. De la sorte, on confine ce programme à certaines *utilisations possibles.*

2.4.4.4 Discussion

Le but premier de DTE est de protéger le système contre le détournement de programmes privilégiés ou l'abus de droits, notamment *via* des « rootkits ». A cette fin, le système offre un mécanisme de définition de politiques strictes, permettant à la fois d'appliquer le principe du moindre privilège et de contraindre dans une certaine mesure le comportement du système.

Le mécanisme de contrôle d'accès de DTE (c'est-à-dire le typage et la DDT) s'applique dans les mêmes conditions que le contrôle d'accès discrétionnaire standard : la légalité de chaque opération exécutée est considérée individuellement et uniquement en fonction du domaine du sujet (processus), du type de l'objet et de la nature de l'opération requise.

Il est important de noter que le simple fait d'exprimer une politique de sécurité dans le modèle DTE ne garantit pas qu'il n'y aura pas de violation. Dans l'exemple de l'attaque exploitant l'imprimante présentée en section 2.2.2.4, la politique de sécurité par défaut est :

- seul le service d'impression *lpd* est autorisé à envoyer des informations à l'imprimante (*/dev/lp0* par exemple) ;
- tous les utilisateurs sont autorisés à transmettre des requêtes à *lpd* uniquement en utilisant le programme *lpr*[15].

Telle quelle, cette politique peut être implémentée dans DTE sous la forme suivante :

- l'imprimante (le périphérique */dev/lp0*) ainsi que la file d'attente de l'impression (le répertoire */var/spool/lp* et ses sous-répertoires) sont de type *printer_t* ;
- le domaine *lpd_t* est autorisé à lire tous les fichiers et à lire/écrire dans les fichiers de type *printer_t* ;
- le domaine *lpr_t* n'est autorisé qu'à écrire dans les fichiers de type *printer_t* ;
- le programme *lpd* est point d'entrée du domaine *lpd_t* ;
- le programme *lpr* est point d'entrée du domaine *lpr_t*.

Dans cette configuration, l'attaque est toujours possible : si l'utilisateur transmet correctement une requête d'impression en utilisant *lpr* et remplace le fichier par un lien symbolique avant que l'impression ne débute réellement, *lpd* imprimera la cible du lien. Pour empêcher effectivement l'attaque, il faudrait interdire la création de liens sur des fichiers auxquels l'accès n'est pas autorisé. Or, DTE ne permet pas d'exprimer une telle règle car elle n'implique pas de lecture ou écriture de fichiers contrôlés. Par ailleurs, ce faisant, on modifirait la sémantique d'une opération du système en prenant en compte la possibilité de l'exploiter dans une attaque, que l'on doit donc préalablement connaître[16].

[15] Cette restriction n'est pas toujours imposée, nous pouvons la mettre en oeuvre afin de protéger le service *lpd* contre d'autres types d'attaques (par exemple par *buffer-overflow*).

[16] On pourrait de même modifier l'opération de création de liens sur un système à con-

Le contrôle de l'exécution, tel qu'il résulte de la DIT, peut en outre donner lieu à des problèmes de définition, principalement la « mainmise » d'un domaine A sur un domaine B : s'il est possible, dans le domaine A, d'écrire dans les points d'entrée d'un domaine B, alors toutes les opérations autorisées dans B deviennent de fait possibles dans A. Il suffit pour cela qu'un processus s'exécutant dans le domaine A remplace un point d'entrée de B par une nouvelle version qui exécute les actions souhaitées. Nous pouvons cependant noter que les cas de « mainmise » peuvent être détectées automatiquement par un utilitaire analysant les fichiers de configuration de DTE.

En conclusion, DTE constitue une plate-forme d'implémentation de politiques à la fois robuste et flexible. De ce point de vue, il complète le contrôle d'accès standard en étendant considérablement la gamme des politiques possibles, mais ne renforce pas la sécurité du système en termes de garantie du respect de la politique.

2.5 Résumé

Le concept de détection d'intrusions paramétrée par la politique constitue une solution prometteuse aux problèmes inhérents aux méthodes de détection classiques, à base de scénarios ou d'anomalies. Deux approches en particulier retiennent notre attention : la spécification de comportement et le contrôle de non-interférence. A nos yeux, toutefois, elles présentent également des inconvénients :

- la détection à base de spécification de comportement exige une connaissance préalable des programmes soumis au contrôle. Si le comportement de certains services (par exemple, un serveur *HTTP*) peut être aisément défini dans les cas simples, la spécification devient un problème difficile dans le cas d'un système complexe. On peut considérer qu'une spécification très fine d'un comportement se ramène *in fine* à une ré-implémentation du programme considéré. Dans ce cas, cette méthode se rapprocherait de fait au modèle de *tolérance aux intrusions* [49, 50], à base de réplication et diversification fonctionnelle. Une telle spécification risque d'être davantage liée au *fonctionnement concret* du logiciel qu'au *service rendu* et ne serait donc pas réutilisable si un logiciel est remplacé par une autre implémentation qui fournit le même service ; elle ne serait pas non plus directement transposable à un autre système ;
- le contrôle de non-interférence de Ko & Redmond ne détecte que la violation de politiques d'intégrité d'une forme particulière et l'implémentation actuelle de cet algorithme introduit des simplifications qui limitent la « couverture » de sa détection.

---

trôle d'accès discrétionnaire standard.

Dans cette thèse, nous avançons qu'un détecteur d'intrusions paramétré par la politique devrait signaler les violations d'une politique *existante et telle qu'elle est définie dans le système*, notamment au moyen des systèmes de contrôle d'accès existants.

Les modèles de contrôle d'accès ont fait l'objet de nombreuses recherches, en particulier dans le domaine de la *vérification de sécurité*. Leur objectif est de permettre la vérification d'une certaine politique et d'offrir des garanties. Malheureusement, les systèmes largement utilisés n'offrent pas de telles possibilités et ne proposent que le simple contrôle d'accès discrétionnaire. Par ailleurs, il peut s'avérer *nécessaire* d'implémenter une politique de contrôle d'accès *connue* pour être non-sûre, pour des besoins pratiques (par exemple, la coopération entre deux utilisateurs *via* des fichiers auxquels les deux peuvent accéder en lecture et écriture), ou du fait des limites inhérentes au système (par exemple, il est actuellement impossible d'autoriser la modification d'une *ligne particulière uniquement* du fichier */etc/shadow*).

Nous pensons donc qu'il existe un besoin pour un système de détection basé sur une politique de contrôle d'accès existante, éventuellement non-sûre. Les violations possibles de sa sûreté devraient être détectées lors de l'exécution.

Le contrôle du flux d'informations fournit un cadre général adapté à cette approche. Néanmoins, le modèle exhaustif de Denning n'est pas applicable dans le cas d'un système d'exploitation classique, sur une architecture matérielle sans facilités de contrôle particulières, et sans informations exhaustives sur les programmes exécutés. Nous devons donc élaborer un modèle de contrôle de flux similaire à ceux utilisés pour assurer la sécurité des applications commerciales, en tirant parti des connaissances que nous avons du système et des flux d'informations *possibles*.

Chapitre 3

Modèle du système

Ce chapitre est consacré à la représentation formelle d'un système, de la politique de sécurité qui lui est appliquée et des violations de celle-ci. Après une discussion intuitive introduisant la classe d'attaques qui fait l'objet de cette étude et son illustration sur plusieurs exemples, nous présentons les hypothèses sur lesquelles est construit notre modèle théorique du système. Nous en déduisons une définition logique d'une violation de la politique de sécurité et l'approche permettant sa détection. Ce modèle a fait l'objet de deux publications [51, 52].

3.1 Attaques par délégation

Dans cette section, nous introduisons informellement la classe d'attaques que nous appelons « attaques par délégation ». Il s'agit de certaines attaques violant une politique de confidentialité et/ou d'intégrité. Nous présenterons trois exemples de telles attaques, puis nous décrirons informellement le mécanisme commun de ces attaques.

3.1.1 Violations de politique de sécurité

Dans cette thèse, nous considérons une intrusion comme une *violation effective des propriétés de la politique de sécurité*. Ceci nous distingue des approches telles que la détection à base de scénarios, qui s'intéressent à *toutes les tentatives de violation*. La classification MITRE/CVE [53] distingue quatre types de violations :

1. *dénis de service*, c'est-à-dire des violations de la propriété de disponibilité ;
2. violations de la propriété de *confidentialité*, c'est-à-dire une politique contrôlant la lecture d'informations ;
3. violations de la propriété d'*intégrité*, c'est-à-dire une politique contrôlant la modification d'informations ;

4. violations de *protection*, qui incluent les cas d'exécution illégale de code, les cas de violation de comportement attendu, les attaques contre l'authentification, etc. De façon générale, il s'agit des violations des *règles* de la politique de sécurité.

Dans cette étude, nous nous concentrons sur les violations *locales* des propriétés de confidentialité et d'intégrité. Nous entendons par violation locale une violation constituée entièrement d'opérations exécutées par le système local. Ceci inclut également certaines attaques initiées par un attaquant distant : par exemple, le fait d'accéder à un fichier confidentiel par l'intermédiaire d'un serveur Web sera considéré comme une violation locale (elle consiste précisément à envoyer le contenu du fichier confidentiel à un client).

L'expérience montre que contrôler la confidentialité et l'intégrité des données en utilisant les méthodes classiques s'avère difficile, pour plusieurs raisons. Étant données des propriétés de confidentialité et d'intégrité, on ne peut pas toujours en constater directement la violation (comme nous le verrons dans l'exemple décrit dans la section 3.1.2.1). D'autre part, il existe un très grand nombre de moyens de contourner une telle politique. Comme le montre la table 3.1, tous les types de vulnérabilités définis par MITRE/CVE peuvent être exploités pour violer la confidentialité et/ou l'intégrité des données[1]. Cependant, dans chaque cas, l'attaque peut également *ne pas violer* la politique et constituer un scénario d'exécution certes inhabituel, mais *légal*. Par exemple, un utilisateur peut potentiellement exploiter par *buffer-overflow* une faille dans un programme privilégié, qu'il utilisera pour lire un fichier dont il est le propriétaire. Dans un tel cas, les alertes levées par un détecteur à base de détection d'anomalies ou de scénarios seront des fausses alertes au regard des propriétés de confidentialité.

3.1.2 Exemple

Les trois exemples suivants illustrent différents cas de violation de politique de confidentialité/intégrité.

3.1.2.1 Attaque contre *lpd*

Nous avons décrit dans la section 2.2.2.4 à la page 51 un exemple d'attaque simple et bien connue contre le service d'impression UNIX original *lpd*. Celui-ci ayant été mis à jour, les systèmes actuels ne sont plus vulnérables à cette attaque, mais le problème est réapparu plus tard dans d'autres services UNIX, par exemple le courrier électronique [54].

[1]Nous ne prenons pas en compte les vulnérabilités du type *erreur de configuration*, car, dans certains cas, on peut les considérer comme des *erreurs de définition des propriétés de la politique*. Par exemple, une erreur de configuration d'un serveur *ftp* peut potentiellement permettre à chacun d'écraser des fichiers existants par l'intermédiaire de la connexion *anonymous*. Dans un tel cas il n'y a pas d'intrusion au sens propre du terme, car les propriétés de la politique autorisent alors cette opération.

	Confidentialité	Intégrité	Protection	Disponibilité
Conditions limites	8.9%	2.6%	4.0%	8.8%
Débordement	3.4%	7.6%	59.2%	28.8%
Validation d'accès	58.1%	53.8%	21.4%	3.4%
Exception	15.8%	6.4%	5.7%	24.7%
Environnement	8.0%	4.6%	3.6%	1.3%
Race condition	2.9%	24.0%	4.3%	1.2%
Conception	*non dispo.*	*non dispo.*	*non dispo.*	*non dispo.*
Configuration	11.3%	11.4%	8.0%	2.9%

TAB. 3.1 – Vulnérabilités vs. violations de politique

L'attaque permet à un utilisateur quelconque d'imprimer le contenu de n'importe quel fichier, même dans le cas où il ne possède pas les droits en lecture. Il s'agit donc d'une violation de la confidentialité du fichier par exploitation de *race-conditions*. On remarquera que cette violation n'a lieu que parce que le fichier est imprimé pour le compte d'un utilisateur qui n'a pas accès au fichier d'après les propriétés de la politique, mais qui a accès à la sortie de l'imprimante.

3.1.2.2 Attaque contre *OpenSSH*

Avec cette attaque reposant sur l'exploitation des bibliothèques partagées au format *ELF* (utilisées, entre autres systèmes, par Linux et Solaris), tout utilisateur peut obtenir les droits de l'administrateur (*root*) [55]. Initialement, l'attaque était utilisée contre le service *telnetd*, ce n'est que plusieurs années après la correction de ce dernier que la même vulnérabilité fut découverte dans *OpenSSH*[2].

Afin d'obtenir une session « *root* », *Alice* procède comme suit :

1. elle écrit une fonction *« setuid »* dont le seul rôle est d'invoquer l'appel système *setuid* original systématiquement avec le paramètre 0, et la compile sous forme d'une bibliothèque partagée *libroot.so* ;
2. dans son profil *ssh*, elle ajoute la déclaration de variable d'environnement suivante :
   LD_PRELOAD=libroot.so ;
3. elle initie une nouvelle session *OpenSSH* ;
4. le service *sshd* lit son profil et affecte les valeurs des variables d'environnement en conséquence ;

[2] Ce qui montre une nouvelle fois l'insuffisance de l'approche « patch-and-pray », où les vulnérabilités sont corrigées ponctuellement en fonction de la connaissance des attaques qui peuvent les exploiter...

5. le service *sshd* exécute le programme *login* afin d'authentifier *Alice* ;
6. la variable *LD_PRELOAD* étant définie, *login* charge la bibliothèque *libroot.so*, ce qui a pour effet de remplacer la fonction *setuid* standard par la version compilée par *Alice* ;
7. après l'authentification, *login* exécute *setuid(uid_alice)* afin de donner à la nouvelle session l'identité d'*Alice*. Du fait de la présence de la bibliothèque *libroot.so*, l'opération réellement exécutée est *setuid(0)* ;
8. Alice obtient ainsi une session avec l'identité 0, c'est-à-dire l'identité de *root*.

Cette attaque réalise une exécution illégale de code, c'est-à-dire *a priori* une violation de *protection*. On peut considérer qu'il s'agit d'une *erreur d'environnement* (en l'occurrence, présence de la bibliothèque *libroot.so*). Du point de vue de la politique, cependant, c'est une violation par Alice de *l'intégrité* du code exécuté par le programme *login*.

3.1.2.3 Attaque contre *Apache*

En exploitant la vulnérabilité décrite dans [56], un attaquant peut exécuter des opérations arbitraires avec les droits du service *httpd*. La procédure précise pour réaliser l'attaque dépend fortement de l'environnement et de la configuration du système cible. Les points principaux en sont les suivants :

1. l'attaquant *Alice* se connecte au service *httpd* du système cible et demande l'accès à une page soumise à authentification ;
2. *Alice* fournit en tant qu'information d'authentification une séquence d'octets particulière, contenant par exemple des séparateurs ;
3. le serveur *Apache* invoque le module *mod_auth_any* afin d'assurer l'authentification ;
4. le module *mod_auth_any* appelle un programme externe en lui transmettant les informations d'authentification fournies par *Alice* ;
5. le module analyse les paramètres. Dans certains cas, la chaîne transmise par *Alice* détourne l'authentification et une partie en est exécutée par exemple en tant que commande du *shell*.

Il s'agit clairement d'une attaque de *protection*. Selon les cas, l'attaquant exploite des erreurs d'*environnement*, de *conditions limites* ou de *validation d'accès*. Toutefois, le scénario d'attaque en lui-même ne constitue pas directement une violation de la politique de confidentialité ou d'intégrité : celle-ci peut se produire ou pas à l'étape 5 selon le code exécuté, par exemple :

- si *Alice* exécute la commande *cat index.html*, il n'y a pas de violation de politique : la lecture de *index.html* par l'intermédiaire du serveur Web est évidemment légale ;

- si *Alice* exécute la commande *cat /home/bob/doc.txt*, il y a violation de la politique de confidentialité qui interdit à *Alice* de lire les documents appartenant à *Bob* ;
- si *Alice* exécute la commande *cat /etc/hostname*, il n'y a pas directement de violation de politique car le fichier */etc/hostname* est généralement accessible en lecture à tout utilisateur. Toutefois, on considérera probablement cette action comme illégale car le contenu de ce fichier n'est pas explicitement publié sur le site Web, en particulier à l'intention des clients distants ;
- etc.

On peut remarquer que bien que l'attaquant peut être distant et connecté *via* TCP, nous considérons cette attaque comme une violation de la politique locale, car le scénario se déroule intégralement sur le hôte cible.

3.1.3 Attaques et flux d'information

D'un point de vue de flux d'informations, une politique de confidentialité ou d'intégrité se traduit par des propriétés de *non-interférence* [22], c'est-à-dire d'*absence de certains flux d'informations.* Si, comme c'est le cas dans les systèmes usuels, la politique est définie en termes de contrôle d'accès des utilisateurs (« sujets ») à des objets, elle interdit de fait certains flux d'informations entre les objets. Par exemple, si l'objet A ne peut être lu que par *Alice* et l'objet B ne peut être écrit que par *Bob*, alors il n'y a pas de flux possible de A vers B.

En généralisant la notion d'objet à tout conteneur d'informations (y compris les « sources » ou les « puits » d'information comme la console ou les sockets réseau), nous pouvons donc modéliser une politique de confidentialité et d'intégrité en termes de flux d'informations autorisés *entre objets.* En considérant que les flux non-autorisés par la politique sont interdits et par conséquent illégaux, une *violation des propriétés de confidentialité/intégrité est un flux d'informations illégal entre objets.*

Nous appelons « attaque par délégation[3] » toute séquence d'opérations qui aboutit à la création d'un flux d'informations illégal entre les objets. Une telle attaque consiste, par hypothèse, en une série d'opérations toutes légales en elles-mêmes (et acceptées en tant que telles par le système d'exploitation, car elles ne violent pas le contrôle d'accès) aboutissant à un résultat illégal sous forme de flux d'informations. Notre notion de *délégation* provient du fait que l'attaquant ne pouvant pas créer directement un tel flux, il doit exploiter l'existence d'un contexte où la création du flux est possible, c'est-à-dire « déléguer » certaines opérations aboutissant à l'attaque. Cette délégation peut prendre diverses formes :

[3] Nous employons ce terme dans un sens différent du sens habituel, où un sujet u *autorise* un autre sujet u' à exécuter certaines opérations pour le compte de u.

- par interférence avec l'exécution d'un processus privilégié, comme dans l'exemple de la section 3.1.2.1. La création du flux illégal (en l'occurrence, le flux depuis un fichier confidentiel vers l'imprimante disponible pour l'utilisateur) est déléguée au processus *lpd* ;
- par construction artificielle d'un tel contexte, comme dans l'exemple 3.1.2.2. *Alice* programme le chargement de *libroot.so* et délègue cette opération au programme d'authentification, ce qui entraîne un flux illégal depuis un objet appartenant à *Alice* (y compris, par exemple, sa console) vers le code en mémoire exécuté par l'authentification ;
- par exploitation de « *confused-deputy* » (programme privilégié dont le fonctionnement est détourné grâce à des paramètres ambigus ou erronés) comme dans l'exemple de la section 3.1.2.3, où le processus *mod_auth_any* et le programme d'authentification exécutent des opérations pour le compte d'*Alice*, alors que ce devraient être des opérations pour le compte du serveur *Apache*. Le flux illégal se produit entre le socket TCP associé au client *Alice* et les objets accèdes finalement par le programme d'authentification ;
- etc.

On remarquera que les scénarios des attaques peuvent être très variés ; toutefois l'attaque sera détectée de manière fiable si l'on contrôle les flux des informations engendrés par les opérations exécutées.

3.2 Représentation du système

Le modèle utilisé pour décrire un système d dans le cadre de cette étude doit se montrer assez indépendant des implémentations réelles pour être applicable à divers systèmes usuels (au moins les systèmes d'exploitation Linux, Unix et Windows), soumis à des politiques de sécurité variées. Simultanément, il doit rester suffisamment réaliste pour permettre la modélisation d'attaques concrètes réelles.

Nous avons retenu une représentation orientée objet du système, avec les hypothèses suivantes que nous détaillerons plus loin :

1. toute information traitée par le système correspond à l'état d'un objet ;
2. l'état d'aucun objet ne peut être lu ni modifié autrement que par l'intermédiaire des méthodes de l'objet ;
3. chaque opération du système correspond à l'invocation d'une ou plusieurs méthode(s) d'un objet ou d'un ensemble d'objets, et/ou à la création ou destruction d'un objet ou d'un ensemble d'objets ;
4. tous les objets du système sont atomiques.

Nous entendons ici par *objet* toute entité identifiée dans le système susceptible de contenir de l'information. L'objet est défini comme un ensemble

d'attributs, dont les valeurs déterminent l'état, et un ensemble de méthodes disponibles pour accéder à ces attributs. Concrètement, les types des objets dépendent du système modélisé. Dans les cas usuels, des exemples d'objets sont les fichiers ou les messages inter-processus. En revanche, nous ne considérons pas en tant qu'objets des entités telles que les processus ou les signaux entre processus, car ils ne contiennent pas à proprement parler des informations soumises à une politique de confidentialité et d'intégrité.

Ainsi, selon la première hypothèse, le modèle ne considère que l'information stockée dans un objet du système sous forme de valeurs de ses attributs. Nous ne prenons pas en compte l'information stockée dans des conteneurs non accessibles explicitement dans le système, tels que les registres du microprocesseur ou la pile. De même, nous ne considérons pas individuellement les bits en mémoire centrale, mais uniquement les pages entières, manipulées effectivement en tant qu'objets par le système et soumises à des règles de contrôle d'accès.

Chaque information possédant un contenant explicite, nous considérons dans la seconde hypothèse que les moyens d'y accéder sont également explicites et dépendent uniquement du contenant. Par exemple, le contenu d'un fichier n'est accessible dans le cadre du modèle que par l'intermédiaire des opérations de lecture/écriture, de même qu'une information en mémoire centrale ne peut être atteinte qu'en « mappant » la page correspondante.

Remarque 3.2.1 *Ces deux hypothèses signifient que nous considérons dans le cadre du modèle que* tous les flux d'informations sont identifiables *en termes d'objets-source, objets-destination et méthodes utilisées sur ces objets. Par conséquent, nous* écartons explicitement le problème des canaux cachés. *Cela réduit naturellement le spectre des attaques détectables dans le cadre de ce modèle ; toutefois, l'expérience montre que les attaques exploitant les canaux cachés demeurent rares et se produisent dans des cas particuliers qui dépassent les objectifs de notre étude.*

La troisième hypothèse constitue un corollaire des deux précédentes. Un système offre un ensemble d'opérations possibles, par exemple des commandes ou des appels systèmes. Dans ce modèle, nous ne considérons que des opérations qui donnent effectivement lieu à des flux d'informations. Par exemple, la lecture d'un fichier crée un flux depuis l'objet « fichier » vers un objet du type « page mémoire » ; mais nous ne considérons pas en tant qu'opération par exemple la création d'un *thread*, car il ne s'agit pas explicitement d'un flux d'informations. D'après les hypothèses 1 et 2, ces flux se traduisent par des appels de méthodes sur les objets concernés. Le fonctionnement du système est ainsi modélisé par des ensembles de flux d'informations, engendrés par l'invocation de méthodes sur les objets.

Afin de ne prendre en compte que les flux d'informations *entre objets* et d'exclure de toute considération des flux internes aux objets, nous formulons la quatrième hypothèse. Selon l'acception généralement admise, un

objet *atomique* est un objet dont l'état forme un tout indivisible, c'est-à-dire, un objet dont il est impossible de modifier la valeur de certains attributs indépendemment des autres. Formellement :

Définition 3.2.2 *Soit o un objet possédant un ensemble M_o de méthodes permettant de modifier son état. Pour chaque $m \in M_o$, notons $w(m)$ le sous-ensemble des attributs de o dont la valeur est modifiée par l'invocation de m. L'objet o est dit* atomique *si* $\forall (m, m') \in {M_o}^2, w(m) \cap w(m') \neq \emptyset$.

Par exemple, un fichier constitue un objet atomique (si nous ne considérons que ses attributs « contenu » et « taille », ils ne peuvent être modifiés individuellement).

Naturellement, l'hypothèse d'atomicité des objets n'est pas complètement réaliste et par conséquent, elle limite l'adéquation du modèle théorique aux systèmes existants (par exemple, une page mémoire n'est en principe pas atomique). Nous l'avons retenue afin de simplifier le modèle ; ce choix se trouve justifié par le fait que les attaques que nous considérons reposent en pratique sur des flux *entre objets*, et le fait qu'il est possible dans certains cas de contourner le problème comme nous le verrons dans le chapitre consacré à l'implémentation.

3.3 Opérations et flux d'informations

En invoquant les méthodes des objets, les opérations génèrent des flux d'informations entre objets. Une séquence d'opérations liées causalement crée un enchaînement de flux, ce qui résulte en une composition des flux.

3.3.1 Opérations élémentaires

Nous avons vu que chaque opération possible sur le système est pleinement déterminée par l'ensemble des objets source et l'ensemble des objets destination, avec les méthodes correspondantes. Dans le cas le plus simple, l'opération consiste en un flux depuis un objet source o_s (par exemple par l'intermédiaire d'une méthode *read*) vers un objet destination o_d (par exemple par l'intermédiaire d'une méthode *write*). Pour exprimer une tel flux, nous utilisons la notation suivante :

$$\{o_s.read\} \gg \{o_d.write\}$$

Des opérations plus complexes font intervenir un ensemble d'objets source et/ou d'objets destination. Par exemple, l'opération POSIX *symlink(2)* de création de liens symboliques lit ses arguments « nom du lien » et « cible du lien » dans des pages mémoire *pnom* et *pptr* et modifie le contenu du répertoire *dir* dans lequel le lien est créé. Si l'on considère toujours que ces

accès se font *via* les méthodes *read* et *write*, cette opération est modélisée par :

$$\{pnom.read, pptr.read\} \gg \{dir.write\}$$

Cette représentation dépend uniquement de la nature de l'opération et reste totalement indépendante de l'état des objets, de même que le rôle de l'appel système *symlink* modélisé ne dépend pas du contenu du fichier cible du lien. D'autre part, on ne modélise ici que le flux d'informations engendré par l'appel-système *symlink*, sans aucune considération quant au succès ou à l'échec de son exécution effective.

3.3.2 Dépendance causale

Chaque opération se traduisant par des flux d'informations, une séquence d'opérations correspond à une succession des flux. Par conséquent, une séquence d'opérations peut, par transitivité, construire un flux d'informations. Suivant la définition de la dépendance causale de Lamport [57] :

Définition 3.3.1 *Il y a dépendance causale entre deux opérations exécutées en séquence si au moins un des objets « destination » de la première opération figure en tant qu'objet « source » dans la seconde opération.*

Dans ce cas, il peut exister un flux d'informations inter-opérations par l'intermédiaire de cet objet ; ainsi, du point de vue des flux générés, la séquence d'opérations :

$$\{o.read\} \gg \{o'.write\}; \{o'.read\} \gg \{o''.write\}$$

est équivalente à l'unique opération :

$$\{o.read\} \gg \{o'.write, o''.write\}$$

En généralisant cette application de la définition de Lamport aux opérations plus complexes, nous pouvons exprimer la règle suivante, illustrée par la figure 3.1 :

Propriété 3.3.2 *Étant données deux opérations* ω_1 *et* ω_2 *de la forme*

ω_1 **:** $\{s_1.ms_1 \dots s_p ms_p\} \gg \{d_1.md_1 \dots d_q.md_q\}$

ω_2 **:** $\{s'_1.ms'_1 \dots s'_u ms'_u\} \gg \{d'_1.md'_1 \dots d'_v.md'_v\}$

notons $o_s(\omega)$ *l'ensemble des objets source de l'opération* ω *et* $o_d(\omega)$ *l'ensemble de ses objets de destination. La séquence* $\omega_1; \omega_2$ *crée des flux entre les mêmes objets que l'ensemble des deux opérations concurrentes* ω'_1 *et* ω'_2 *:*

ω'_1 **:** $\{s_1.ms_1 \dots s_p.ms_p, s'_i.ms'_i\} \gg \{d_j.m_j, d'_1.md'_1 \dots d'_v.md'_v\}$ *où* $i \in [1 \dots u] \mid s'_i \notin o_d(\omega_1)$ *et* $j \in [1 \dots q] \mid d_j \in o_s(\omega_2)$

ω'_2 **:** $\{s_1.ms_1 \dots s_p.ms_p\} \gg \{d_k.md_k\}$ *où* $k \in [1 \dots q] \mid d_k \notin o_s(\omega_2)$

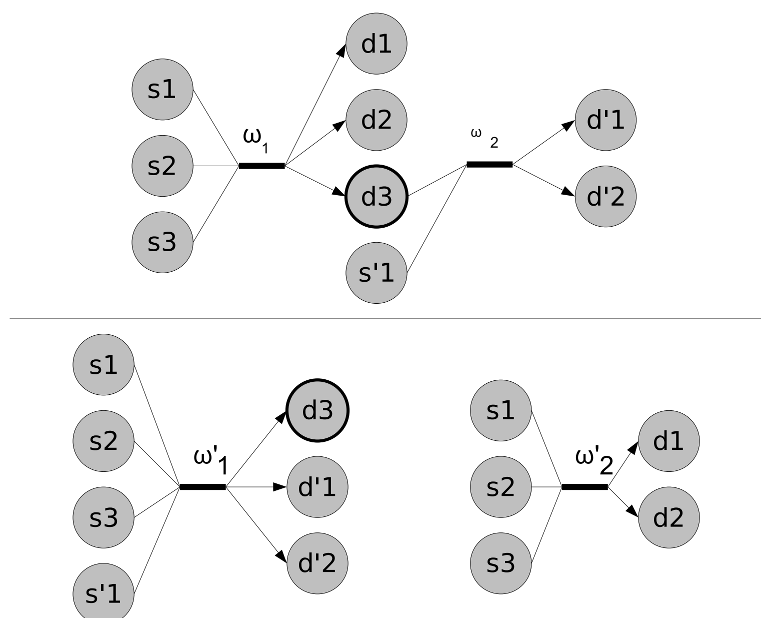


FIG. 3.1 – Exemple le décomposition de flux

Naturellement, cette propriété concerne uniquement le schéma des flux entre objets et non l'information transférée elle-même. On observera que cette transformation d'une séquence de deux opérations avec dépendance causale en deux opérations concurrentes est possible grâce à l'hypothèse d'atomicité des objets. En effet, cette dernière garantit que la règle de dépendance causale s'applique indépendemment des méthodes utilisées.

3.4 Politique de sécurité et domaines

En traduisant la politique de confidentialité/intégrité sous forme de flux autorisés, nous définissons des ensembles de flux autorisés appelés *domaines*.

3.4.1 Interprétation de la politique de sécurité

Étant donné un ensemble d'objets, une politique de confidentialité et d'intégrité définie sous forme de règles de contrôle d'accès autorise de fait certains flux d'informations entre objets.

3.4.1.1 Exemple 1 : contrôle d'accès discrétionnaire

Nous décrirons précisément l'interprétation d'une politique de type DAC dans le chapitre n°4 consacré à l'implémentation ; nous ne traiterons ici qu'un exemple simplifié destiné à illustrer notre propos.

Étant donnée par exemple la politique de contrôle d'accès discrétionnaire présentée dans la table de la figure 3.2, *Alice* est en mesure de transférer librement le contenu de o_1 vers o_2 et o_3. *Bob* peut transférer les contenus de o_2 et o_4 vers o_1, de même que le contenu de o_2 vers o_4. Le schéma de la figure 3.2 récapitule les flux autorisés entre ces objets.

Par définition, *Alice* peut librement invoquer toutes les méthodes autorisées ; tous les flux d'informations entre objets construits à l'aide de ces méthodes sont donc légaux. De même, les appels de méthodes autorisés de *Bob* donnent lieu à un ensemble de flux d'informations légaux.

Chacun de ces ensembles constitue une *fermeture transitive*. En effet, *Bob* peut, par exemple, exécuter soit la séquence d'opérations

$$\{o_2.lecture\} \gg \{o_4.ecriture\}; \{o_4.lecture\} \gg \{o_1.ecriture\},$$

soit directement l'opération

$$\{o_2.lecture\} \gg \{o_1.ecriture\},$$

en aboutissant finalement dans les deux cas à un flux depuis o_2 vers o_4 comme l'indique la propriété 3.3.2. De même, rien n'interdit *a priori* de combiner les appels de méthodes en des opérations plus complexes, comme

$$\{o_2.lecture, o_4.lecture\} \gg \{o_1.ecriture\}.$$

Les deux ensembles sont *exclusifs*. Par exemple, la combinaison d'un flux de o_1 vers o_2, légal pour *Alice*, avec un flux depuis o_2 vers o_4, légal pour *Bob*, donne lieu à un flux depuis o_1 vers o_4. Or un tel flux ne fait pas partie des flux autorisés par la politique de sécurité ; la combinaison est donc interdite.

3.4.1.2 Exemple 2 : muraille de Chine

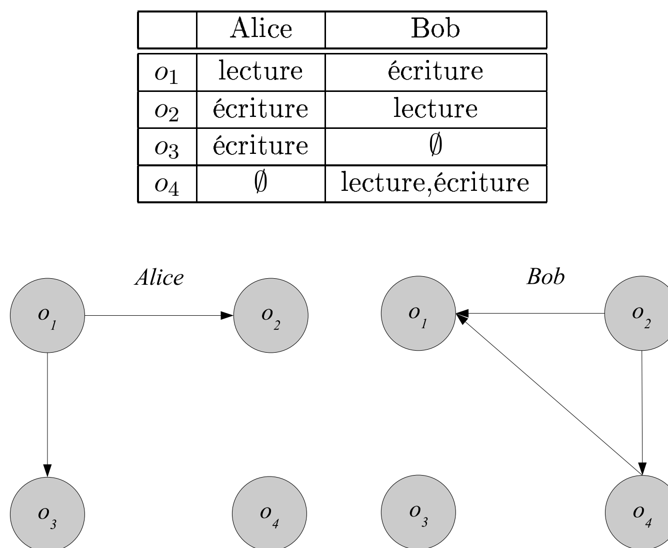
Une politique du type muraille de Chine simple (voir section 2.3.2.3 du chapitre 2) consiste en :

- des classes de conflit d'intérêts $C_1, C_2 ... C_n$;
- au sein de chaque classe de données C_i, des ensembles de données :

$$d_{i,1}, d_{i,2} ... d_{i,p_i}$$

Soit o_1 un objet appartenant à une classe de conflit d'intérêts C_i et un ensemble de données $d_{i,j}$. De même, soit o_2 un objet appartenant à une classe de conflit d'intérêts $C_{i'}$ et un ensemble de données $d_{i',j'}$. Un flux de o_1 vers o_2 est légal dans deux cas :

1. si o_1 et o_2 appartiennent à un même ensemble de données, c'est-à-dire si $i = i'$ et $j = j'$;
2. si o_1 et o_2 appartiennent à deux classes distinctes, c'est-à-dire si $i \neq i'$.



	Alice	Bob
o_1	lecture	écriture
o_2	écriture	lecture
o_3	écriture	$\emptyset$
o_4	$\emptyset$	lecture,écriture

FIG. 3.2 – Interprétation du contrôle d'accès

3.4.2 Domaines

En exprimant une politique de sécurité sous forme de flux d'informations autorisés entre objets, nous obtenons donc des ensembles de flux légaux. Nous appelons ces ensembles des *domaines*.

Définition 3.4.1 *Un* domaine *est la fermeture transitive d'un ensemble de flux légaux du point de vue de la politique de sécurité.*

D'après cette définition, il est évident que toute combinaison de flux appartenant à un même domaine est légale ; toute combinaison de flux appartenant à des domaines distincts est illégale.

A titre d'exemple, dans le cas de la simple politique discrétionnaire de la figure 3.2, les domaines correspondent directement aux actions possibles des utilisateurs *Alice* et *Bob* (cf. figure 3.3).

D'après les hypothèses établies dans la section 3.2, chaque flux d'informations suppose l'invocation de méthodes d'au moins deux objets (un objet source, un objet destination). En tant qu'ensemble de flux, un domaine peut donc être modélisé par un ensemble de couples « objet,méthode ».

Dans l'exemple de la politique que nous considérons (figure 3.2), ces ensembles correspondent donc aux capacités des deux utilisateurs *Alice* et *Bob* :

- le premier domaine comprend la méthode *lecture* de l'objet o_1 ainsi que la méthode *écriture* des objets o_2 et o_3 ;

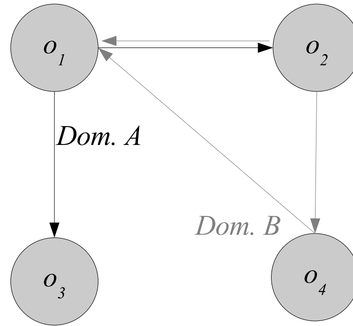


FIG. 3.3 – Domaines correspondant à la politique

– le second domaine comprend les méthodes *lecture* et *écriture* de o_4, ainsi que respectivement les méthodes *lecture* et *écriture* de o_2 et o_1.

Par définition, tout flux d'informations créé par un ensemble d'invocations de méthodes sur des objets associées à un même domaine appartient à ce domaine, et tout flux appartenant à un domaine peut être créé par un ensemble d'invocations de méthodes associées à ce domaine. Il en résulte la *règle d'unicité de domaine* :

Définition 3.4.2 *Un flux d'informations est légal si et seulement tous les couples (méthode,objet) nécessaires à la création de ce flux sont associés à un même domaine.*

Par exemple, la méthode *lecture* de o_1 n'est associée qu'au premier domaine et la méthode *écriture* de o_4 n'est associée qu'au second domaine, un flux d'informations créé par l'utilisation de ces deux méthodes n'est donc pas légal.

3.5 Modèle formel

A présent, nous définissons formellement le modèle du système, la notion d'attaque par délégation et la méthode pour détecter ces attaques.

3.5.1 Références

L'ensemble des objets et appels de méthode utilisés par une opération ne dépend pas de l'état de ces objets mais uniquement de la nature de cette opération. Le seul critère de légalité d'un flux étant la règle d'unicité de domaine, il est par conséquent inutile de prendre en compte l'*état* des objets (c'est-à-dire l'information contenue dans les objets) dans le modèle. Pour

contrôler les flux entre objets, seule la modélisation des domaines est nécessaire. A cette fin, nous introduisons la notion de *référence* :

Définition 3.5.1 *Étant donné un domaine d, un objet o et une méthode m, la* référence *$R_d o.m$ représente la capacité de créer des flux d'informations appartenant au domaine d en utilisant la méthode m de l'objet o.*

A chaque objet est ainsi associé un ensemble de références, déterminant dans quel(s) domaine(s) chaque méthode de l'objet peut être utilisée.

Nous représentons donc un système par l'ensemble de ses objets, l'ensemble des méthodes, l'ensemble des opérations disponibles qui exploitent ces méthodes, l'ensemble des domaines définis et une fonction que nous appellerons *fonction d'état* qui associe à chaque objet un ensemble de références :

Définition 3.5.2 *Un système est un triplet (O, M, Ω) où :*
- *O est l'ensemble des identificateurs d'objets ;*
- *M est l'ensemble des méthodes d'objets ;*
- *$\Omega \subseteq \{\mathcal{P}(O \times M) \gg \mathcal{P}(O \times M)\}$ est l'ensemble des opérations qui génèrent des flux entre objets. La notation $\mathcal{P}(A)$ désigne l'ensemble des n-uplets composés d'éléments de l'ensemble A ;*

Une politique de sécurité est un couple (D, s) où :
- *D est l'ensemble des identificateurs de domaines ;*
- *$s \in S$ est une fonction d'état définissant l'ensemble des références associées à chaque objet, S étant l'ensemble des fonctions de la forme*

$$O \rightarrow \mathcal{P}(\{R_d o.m\}, d \in D, o \in O, m \in M)$$

C'est-à-dire qu'étant donnée une fonction d'état $s \in S$ et un objet $o \in O$, $s(o)$ est l'ensemble de références associées à s dans l'état s.

On peut noter que les ensembles O, M et Ω décrivent le système sous forme d'objets et de flux possibles entre les objets, tandis que l'ensemble D et la fonction s implémentent une politique de sécurité en définissant les flux d'informations légaux.

3.5.2 Légalité des opérations et règle d'unicité

Par définition, un flux est légal s'il appartient à un domaine. La règle d'unicité de domaine exprimée dans la section 3.4.2 impose que tous les appels de méthode nécessaires à la création du flux soient possible dans un même domaine. Par conséquent, étant donné une fonction d'état du système, le flux est légal si les ensembles de références disponibles pour les appels de méthodes requis ont au moins un domaine commun.

Appelons $dom(o_1.m_1 \ldots o_p.m_p, s)$ l'ensemble des domaines dans lesquels les appels de méthode $o_1.m_1$ à $o_p.m_p$ sont possibles selon la fonction d'état s :

Définition 3.5.3 *Étant donné* p *appels de méthodes* $o_1.m_1 \dots o_p.m_p$,

$$dom(o_1.m_1 \dots o_p.m_p, s) = \{d \in D \mid \forall i \in [1 \dots p], R_d o_i.m_i \in s(o_i)\}$$

Nous pouvons formellement exprimer la légalité d'une opération par le prédicat $islegal : \Omega \times S \rightarrow \{vrai, faux\}$ suivant :

Étant donné une opération ω de la forme

$$\{o_1.m_1 \dots o_p.m_p\} \gg \{o'_1.m'_1 \dots o'_q.m'_q\}$$

et une fonction d'état s, $islegal(\omega, s) = vrai$ si et seulement si :

$$dom(o_1.m_1 \dots o_p.m_p, s) \cap dom(o'_1.m'_1 \dots o'_q.m'_q, s) \neq \emptyset$$

Il s'agit d'une retranscription formelle directe de la règle d'unicité de domaine.

Remarque 3.5.4 *Dans ce modèle, la création et destruction d'objets n'est pas représentée en tant que telle. Les opérations qui créent de nouveaux objets (par exemple, l'opération* $creat(2)$ *de l'API POSIX) peuvent être modélisées simplement en tant que flux où l'objet créé figure en tant qu'objet de destination.*

3.5.3 Flux de références

Une séquence de deux opérations causalement dépendantes donne lieu à des flux d'informations composés. La séquence est donc par définition légale, si et seulement si ces flux respectent la règle d'unicité de domaine.

Dans l'exemple de la figure 3.3, la séquence

$$\{o_1.lecture\} \gg \{o_2.ecriture\}; \{o_2.lecture\} \gg \{o_4.ecriture\}$$

équivaut, d'après la propriété 3.3.2, au flux

$$\{o_1.lecture\} \gg \{o_2.ecriture, o_4.ecriture\}$$

La séquence est donc légale si, et seulement si, ce dernier flux est légal. Par conséquent, l'exécution de la seconde opération aurait été légale si et seulement si le flux $\{o_1.lecture\} \gg \{o_2.ecriture, o_4.ecriture\}$ avait été légal, ce qui n'est pas le cas. L'exécution de la première opération donne par conséquent lieu à un *nouvel état du système*, dans lequel la seconde opération est désormais *interdite*.

La généralisation de ce raisonnement s'appuie sur le principe qu'un flux d'informations n'est pas légal s'il donne lieu, par composition, à la création d'un flux illégal. Par conséquent, lorsqu'un flux entre objets se produit, les opérations (et donc les flux) désormais possibles sur les objets de destination doivent être équivalentes à celles initialement possibles sur les objets source :

cela implique que les ensembles de références associés finalement aux objets de destination doivent être équivalents à ceux associés initialement aux objets source.

Un flux d'information entre objets sera donc représenté par un *flux de références* des objets source vers les objets destination. Nous le définissons formellement par la fonction *exec* :

Définition 3.5.5 *exec est une fonction $\Omega \times S \rightarrow S$ telle qu'étant donnée une opération ω de la forme $src \gg dst$, src et dst étant deux ensembles d'invocations de méthodes sur des objets, et une fonction d'état s, $exec(\omega, s)$ est une fonction d'état s' telle que $\forall o \in O$:*

- si $\exists m \in M | o.m \in dst$ alors $s'(o) = \{R_d o.m' \text{tel que } d \in dom(src, s) \text{ et } \exists o' \in O, m'' \in M \text{ tels que } o'.m'' \in src \text{et } R_d o'.m' \in s(o')\}$
- sinon $s'(o) = s(o)$.

Cette fonction construit, pour chaque objet de destination, un ensemble de références autorisant l'invocation des méthodes identiques à celles possibles sur les objets source, dans le(s) domaine(s) où l'opération exécutée est légale.

En utilisant la condition de légalité des opérations (déf. 3.5.2) et la fonction *exec*, nous pouvons facilement définir la légalité d'une séquence de deux opérations :

Théorème 3.5.6 *Étant donné une fonction d'état s et deux opérations légales ω_1 et ω_2, la séquence $\omega_1; \omega_2$ est légale si et seulement si :*

$$islegal(\omega_1, s) = vrai \ et \ islegal\left[\omega_2, exec(\omega_1, s)\right] = vrai$$

Preuve :

Notons (a) la propriété du théorème. Soit s une fonction d'état. Soient ω_1 et ω_2 deux opérations légales :

$$\omega_1 : \{os_1.ms_1 \ldots os_p ms_p\} \gg \{od_1.md_1 \ldots od_q.md_q\}$$
$$\omega_2 : \{os'_1.ms'_1 \ldots os'_u.ms'_u\} \gg \{od'_1.md'_1 \ldots od'_v.md'_v\}$$

Notons $src(\omega)$ et $dst(\omega)$ les ensembles des invocations de méthodes source et destination respectifs d'une opération ω.

- Si ω_1 et ω_2 ne sont pas causalement dépendantes, c'est-à-dire si quels que soient o et m tels que $o.m \in src(\omega_2)$ et quel que soit m', $o.m' \notin dst(\omega_1)$, alors d'après la définition de *exec* :

  $$\forall o.m \in src(\omega_2), s(o) = exec(\omega_1, s)(o)$$

  Dans ce cas, (a) équivaut donc à :

  $$islegal(\omega_1, s) = vrai \text{ et } islegal(\omega_2, s) = vrai$$

  Ceci est vrai par hypothèse, donc le théorème est valide dans ce cas.

– Si ω_1 et ω_2 sont causalement dépendantes, alors d'après la propriété 3.3.2, les flux entre objets créés par la séquence $\omega_1;\omega_2$ sont similaires à ceux créés par les des deux opérations concurrentes :

ω'_1 : $\{s_1.ms_1 \dots s_p.ms_p, s'_i.ms'_i\} \gg \{d_j.m_j, d'_1.md'_1 \dots d'_v.md'_v\}$ où $i \in [1 \dots u] \mid s'_i \notin o_d(\omega_1)$ et $j \in [1 \dots q] \mid d_j \in o_s(\omega_2)$

ω'_2 : $\{s_1.ms_1 \dots s_p.ms_p\} \gg \{d_k.md_k\}$ où $k \in [1 \dots q] \mid d_k \notin o_s(\omega_2)$

La séquence $\omega_1;\omega_2$ est donc légale si et seulement si les deux opérations ω'_1 et ω'_2 sont légales, c'est-à-dire :

$$islegal(\omega'_1, s) = vrai \text{ et } islegal(\omega'_2, s) = vrai \ (b)$$

D'après la définition de $islegal$ (cf. déf. 3.5.2), (b) équivaut à :

1. $dom[src(\omega'_1), s] \cap dom[dst(\omega'_1)] \neq \emptyset$; et
2. $dom[src(\omega'_2), s] \cap dom[dst(\omega'_2)] \neq \emptyset$.

D'après la définition de dom (cf. déf. 3.5.3), on a trivialement :

$$dom(A) \cap dom(B) = dom(A \cup B)$$

donc (b) équivaut à :

1. $dom\,[src(\omega'_1) \cup dst(\omega'_1), s] \neq \emptyset$; et
2. $dom\,[src(\omega'_2) \cup dst(\omega'_2), s] \neq \emptyset$

Or, $src(\omega'_1) = src(\omega_1) \cup src(\omega_2) - \{od_i.md_i | od_i \in o_d(\omega_1)\}$ et $dst(\omega'_1) = dst(\omega_1) \cup dst(\omega_2)$. D'autre part, $src(\omega'_2) = src(\omega_1)$ et $dst(\omega'_2) = dst(\omega_1) - \{os_j.md_j | os_j \in o_s(\omega_2)\}$. Par conséquent, (b) équivaut à :
hypothèse, $d_i.md_i = s'_j.ms'_j$, donc (b) équivaut à :

1. $dom\,[src(\omega_1) \cup src(\omega_2) \cup dst(\omega_2), s] \neq \emptyset$; et
2. $dom\,[src(\omega_1) \cup dst(\omega_1) - \{os_j.md_j | os_j \in o_s(\omega_2)\}, s] \neq \emptyset$

Si ω_1 est légale, la seconde proposition est trivialement vraie, donc (b) équivaut à :

1. $dom\,[src(\omega_1) \cup src(\omega_2) \cup dst(\omega_2), s] \neq \emptyset$; et
2. $islegal(\omega_1, s) = vrai$

D'après la définition de $exec$, il découle directement que

$$dom\,[src(\omega_1) \cup src(\omega_2) \cup dst(\omega_2), s] = dom\,[src(\omega_2) \cup dst(\omega_2), exec(\omega_1, s)]$$

Par conséquent, (b) équivaut à :

1. $dom\,[src(\omega_2) \cup dst(\omega_2), exec(\omega_1, s)] \neq \emptyset$
2. $islegal(\omega_1, s) = vrai$

Ceci est la propriété (a). Or, (b) dont nous venons de montrer l'équivalence à (a) est la condition de légalité de la séquence $\omega_1;\omega_2$, donc la validité du théorème est démontrée. $\square$

L'interprétation intuitive de la fonction *exec* et du théorème 3.5.6 consiste à considérer les références comme un *contrôle d'accès au contenu*, par opposition au *contrôle d'accès au contenant* habituel. Dans le système de la figure 3.3, l'application de la fonction *exec* à l'opération $\{o_1.lecture\} \gg \{o_2.ecriture\}$ crée l'état du système de la figure 3.4, où les flux de o_2 vers o_4 ne sont plus légaux. En effet, la création d'un tel flux serait équivalente, par composition, à un flux de o_1 vers o_4 qui n'est pas autorisé par la politique de sécurité.

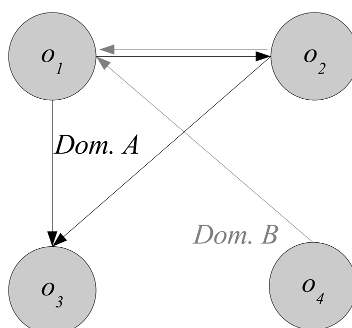


FIG. 3.4 – Résultat du flux de o_1 vers o_2

Remarque 3.5.7 *La définition de la fonction exec suppose implicitement que les objets source et destination possèdent les mêmes méthodes, c'est-à-dire que ce sont des objets de même classe. Les systèmes actuels, en particulier ceux de type UNIX, tendent en effet à imposer la même interface de type fichier à tous les objets. Cependant, cette supposition ne remet nullement en cause la généralité du modèle que nous proposons car, si les objets source et destination sont de classes différentes, deux cas peuvent se présenter :*

- *les références propagées aux objets de destination par la fonction exec portent sur des méthodes que ces objets ne possèdent pas. Aucun flux utilisant ces références n'aura donc lieu, mais leur existence ne compromet pas la sécurité du système ;*
- *l'accès ultérieur aux objets de destination nécessite d'utiliser des invocations de méthodes autres que celles accessibles par les références propagées. Ce problème est aisément résolu par l'utilisation d'objets « virtuels » proposant ces références. Il s'agit en réalité d'un problème*

d'adaptation du modèle à un système réel et sera décrit en détail dans le chapitre 4, consacré à l'implémentation du modèle.

3.6 Détection d'intrusions

Nous avons vu dans la section 3.1.3 qu'une attaque par délégation se traduit par un flux d'informations illégal entre objets, c'est-à-dire un flux nécessitant des références appartenant à plusieurs domaines distincts.

Un scénario d'attaque consiste en une séquence d'opérations (c'est-à-dire une *trace*). La politique de sécurité, définie par un ensemble de domaines et une fonction d'état initiale s_0, est violée par la trace si :

- la trace comprend une opération créant un flux illégal ; ou
- la succession des opérations de la trace crée un flux illégal par composition.

Par l'application itérative du théorème 3.5.6, nous obtenons l'algorithme de vérification de trace 1.

Algorithm 1 Vérification de légalité d'une trace

Données :

- un système (O, M, Ω) ;
- une politique de sécurité (D, s) ;
- une trace $\omega_1 \dots \omega_n \in \Omega^n$

Vérification : soit $s_0 = s$, s étant la fonction d'état définie par la politique de sécurité.

Pour tout i de 1 à n :

1. si $islegal(\omega_i, s_{i-1}) = faux$
   alors signaler ω_i en tant que violation de la politique
2. $s_i = exec(\omega_i, s_{i-1})$

La signalisation des opérations violant la politique est indépendante du modèle et de l'algorithme de vérification.

- la solution la plus simple consiste à lever une *alerte*, tandis que l'exécution continue sans interruption. Il s'agit de la forme la plus courante actuellement de détection d'intrusions ;
- l'opération peut être autorisée sans que ses effets ne soient effectivement répercutés sur les objets concernés, dans un but de *tolérance aux intrusions* ou de « *honeypotting* » ;
- l'opération et les flux ultérieurs causalement dépendants peuvent être stockés dans un journal, afin de permettre l'analyse légiste (« *forensics analysis* ») *a posteriori*. Bien qu'attrayante, cette solution est difficilement réalisable telle quelle, en raison du volume de données à traiter et de son l'impact considérable sur les performances du système.

3.7 Résumé

Nous avons proposé un formalisme permettant de modéliser un système d'exploitation usuel sous forme de flux d'informations et une politique de sécurité sous forme de flux légaux. Nous avons alors établi un algorithme de détection des violations de cette politique, basé sur le contrôle des dépendances causales des flux.

Nous présentons dans l'annexe B une comparaison avec l'approche publiée par Ko et Redmond dans [21], en montrant que les intrusions détectables avec cette dernière le sont également en utilisant l'approche que nous proposons.

Chapitre 4

Implémentation

Afin d'évaluer l'utilisabilité et l'efficacité pratique de l'approche que nous proposons, nous avons réalisé un prototype de détecteur d'intrusions basé sur le flux de références. Dans ce chapitre, nous décrivons la problématique générale de l'implémentation du modèle et les solutions que nous avons retenues. Les expériences réalisées à l'aide de ce prototype et les résultats obtenus sont présentés dans le chapitre 5.

Notre prototype fonctionne sur le système d'exploitation Linux. En effet, l'ouverture du système facilite considérablement l'implémentation de nouvelles fonctionnalités. Grâce à des outils tels que *Usermode Linux* [58], il est aisé de développer, tester et déboguer des modules du noyau. En outre, nous verrons que si l'on ignore les canaux cachés, comme nous en avons fait le choix, alors l'architecture du système se montre bien adaptée au modèle théorique tel que nous l'avons développé dans le chapitre 3. Finalement, Linux est un système largement répandu, utilisé pour exploiter de nombreuses applications. En même temps, il est la cible de nombreuses attaques [59]. En tant que tel, ce système nous fournit donc un environnement de tests réaliste.

Dans ce chapitre, nous abordons successivement l'adaptation de notre modèle à l'architecture de Linux et la représentation sous forme de références et domaines des politiques de confidentialité et d'intégrité applicables à Linux. Nous présentons ensuite les aspects techniques de notre implémentation. Enfin, nous dressons un bilan de cette démarche et nous discutons de la problématique liée à la réalisation d'un logiciel similaire sur d'autres systèmes d'exploitation.

4.1 Modèle du système Linux

Afin de mettre en oeuvre notre approche sous Linux, nous devons représenter ce système sous forme d'ensembles d'objets, de méthodes et d'opérations engendrant des flux d'information (respectivement les ensembles O, M et Ω

du modèle théorique présenté dans le chapitre 3).

4.1.1 Objets et méthodes

Nous avons défini un *objet* en tant qu'entité identifiée dans le système et contenant de l'information. Linux suit, à quelques exceptions près, l'architecture originale d'Unix, où tout conteneur d'information est assimilé à un fichier. Dans le cas de notre modèle, cette conception présente l'avantage d'une grande cohérence. L'ensemble de méthodes à prendre en compte est très réduit et contient essentiellement les deux méthodes permettant respectivement de consulter et de modifier le contenu d'un objet : *read* et *write*.

La table 4.1 liste les différentes classes d'objets existant dans Linux et les méthodes correspondantes. Nous verrons dans la section 4.1.1.6 que les attaques que nous considérons ne nécessitent pas de prendre en compte les périphériques et les variables du noyau.

classe d'objets	méthodes
fichier,tube,fifo,autre descripteur E/S	read,write
attribut de fichier	read,write
socket	read,write,accept
page mémoire	map_r,map_w
espace d'adressage	read,write
message inter-processus	read,write
périphérique	*non pris en compte*
variable du noyau	*non pris en compte*

TAB. 4.1 – Objets et méthodes du système Linux

4.1.1.1 Fichiers et descripteurs d'E/S

La plupart des transferts d'information sous Linux se font par l'intermédiaire des descripteurs d'entrées/sorties. Ces descripteurs ne figurent pas en tant que tels dans le modèle, ils ne constituent pas des objets. En effet, les opérations sur plusieurs descripteurs différents peuvent en réalité concerner la même information, de même qu'un même descripteur peut successivement désigner des informations différents (par exemple, en cas de fermeture d'un descripteur puis de sa réouverture sur un autre fichier).

Chaque lecture ou écriture sur un descripteur est par conséquent interprétée comme l'invocation d'une méthode *read* ou *write* sur l'objet auquel ce descripteur est lié. Nous pouvons distinguer deux cas :

1. les *fichiers* et *répertoires* : il s'agit d'une invocation directe de méthode sur le fichier ou le répertoire en question ;

2. les *tubes*, les *fifos* et les *descripteurs par défaut (stdin, stdout, stderr)* : la méthode est invoquée sur un objet virtuel représentant le tampon auquel ce descripteur est lié.

Dans le second cas, les informations transitant par les descripteurs ne sont pas stockées dans un objet explicitement nommé, les descripteurs sont alors les seuls identificateurs permettant d'y accéder. Dans le modèle, nous utilisons alors un objet virtuel représentant le conteneur de ces informations.

Les méthodes *read* et *write* sont les seules que nous ayons besoin de prendre en compte ici. En effet, grâce à l'hypothèse d'*atomicité des objets* (voir chapitre 3, section 3.2), le contenu d'un fichier (ou d'un objet virtuel) est considéré en tant qu'un tout indivisible, qu'il est possible que de consulter ou modifier dans sa totalité. Nous assimilons alors des opérations telles que l'ajout au fichier ou la modification d'un fichier existant sans écraser totalement son contenu à de simples lectures et écritures, tandis qu'un utilitaire tel que *touch* réalise une ouverture de fichier suivie immédiatement de sa fermeture, sans lecture ni écriture.

Strictement parlant, l'objet atomique « fichier » est constitué du contenu et des trois attributs « date » associés (création, dernière modification et dernier accès) . Toutefois, nous pouvons considérer que les dates ne fournissent pas réellement un canal de transfert d'information explicite. Aussi, nous pouvons les considérer comme des canaux cachés et en tant que tels, ils ne sont pas pris en compte dans le modèle.

4.1.1.2 Attributs des fichiers

Les attributs des fichiers autre que les trois dates constituent des objets atomiques à part entière : en effet, l'accès à leur contenu est indépendant de l'accès au contenu du fichier. Ces attributs sont :

- le propriétaire et le groupe auquel appartient le fichier ;
- les permissions et les listes de contrôle d'accès (*ACL*) ;
- les attributs librement définissables par l'utilisateur (par exemple, le type *MIME*, la date de la dernière sauvegarde ,etc.).

De même que nous ne tenons pas compte des dates, nous ne considérons pas les consultations et les modifications des attributs liés au contrôle d'accès (propriétaire, groupe, bits de permissions et ACL). En revanche, les valeurs de ces attributs sont utilisés pour la construction automatique des domaines, comme nous le verrons dans la section 4.2.

Seuls entrent donc en compte les attributs librement définissables, chacun étant un objet atomique indépendant. A nouveau, seule la lecture et la modification de la valeur de l'attribut est possible, donc seules les méthodes *read* et *write* sont définies.

4.1.1.3 Sockets

En tant que descripteurs d'E/S, les sockets sont modélisées avant tout sous forme d'objets virtuels, possédant les méthodes *read* et *write*.

Dans le cas de transferts locaux d'informations au travers des sockets (par exemple, des sockets purement locales, ou des sockets *TCP* et *UDP* où le serveur et le client se trouvent sur le même système), les règles de dépendance causale s'appliquent normalement. La situation est différente pour les sockets correspondant à une connexion distante : ne pouvant faire aucune hypothèse quant aux flux d'informations produits par le système distant, nous devons ignorer la dépendance causale sur ces objets virtuels et considérer chaque transfert indépendemment. Notre implémentation n'est donc pas en mesure de détecter des attaques distribuées.

Outre la lecture et l'écriture, les sockets disposent de la méthode *accept* qui correspond à la connexion d'un client. En effet, une politique de sécurité peut réglementer les flux d'informations possibles depuis ou vers les sockets d'un service particulier (par exemple, HTTP). Or, toute connexion d'un client sur la socket primaire d'un service donne automatiquement lieu à la création d'une socket secondaire, utilisée par la session entre le client et le serveur. La politique de sécurité doit donc s'appliquer aux flux depuis ou vers cette *socket secondaire*. Par conséquent, ces flux doivent être traités comme la conséquence causale de la connexion du client sur la socket primaire :

Propriété 4.1.1 *Étant données une socket primaire p et une socket secondaire s, la connection d'un client sur p donnant lieu à la création de s est modélisée par le flux*

$$\{p.accept\} \gg \{s.write\}$$

Les sockets étant par nature des objets à accès séquentiel, nous pouvons considérer que chaque accès à une socket est une modification de l'état de celle-ci. Les sockets sont ainsi assimilés à des objets atomiques.

4.1.1.4 Mémoire virtuelle

Théoriquement, les pages mémoire sont les seuls objets dans Linux qui violent la définition d'un objet atomique. Chaque octet d'une page mémoire peut naturellement être modifié indépendemment des autres. Une page de n octets devrait donc en principe être modélisée en tant qu'objet possédant les méthodes $write_byte_1$, $read_byte_1$, ..., $write_byte_n$, $read_byte_n$, ce qui serait équivalent à considérer chaque octet mémoire en tant qu'objet indépendant.

Dans ce cas, une analyse statique du code source ou une connaissance préalable des programmes serait toutefois nécessaire afin de pouvoir tenir compte des opérations de transfert de données élémentaires au niveau du

microprocesseur, comme nous l'avons vu lors de la discussion sur les méthodes générales de contrôle de flux d'informations (voir section 2.3.3.2 du chapitre 2). Toutefois, de même que les fichiers peuvent être assimilés à des objets atomiques, le contenu d'une page forme également un tout. La lecture d'une partie de la page est un accès au contenu de celle-ci et la modification d'au moins un octet de la page est une modification de l'état de celle-ci. Le mapping d'une page dans un espace d'adressage consiste alors en un flux d'informations entre la page et l'espace d'adressage concerné. De fait, les espaces d'adressage (par exemple, des tampons partagés suivant l'API *IPC POSIX* ou *System V*, ou les espaces *text* et *data* d'un processus en cours d'exécution) apparaissent comme comme des objets atomiques, avec les méthodes habituelles *read* et *write*. En tant que tel, le flux d'informations produit par un mapping doit respecter la règle d'unicité de domaine. Les pages mémoire possèdent donc deux méthodes relatives au mapping :

- *map_r* : mapping en lecture ;
- *map_w* : mapping en écriture.

Nous utilisons ces méthodes spécifiques pour les pages au lieu des méthodes courantes *read* et *write*, car Linux ne permet pas d'accéder à une page directement. Une page n'est accessible que si elle est préalablement « mappée » dans un espace d'adressage.

Remarque 4.1.2 *Sous Linux, le contrôle d'accès à la mémoire s'applique précisément aux pages et aux espaces d'adressage, qui sont donc considérés comme des objets « élémentaires » de ce point de vue.*

4.1.1.5 Messages entre processus

Un message (*IPC System V* ou *POSIX*) est naturellement modélisé par un objet atomique virtuel très simple. L'émission du message correspond à la définition de l'état de cet objet et la réception du message à sa consultation. Les deux méthodes disponibles sont donc encore *read* et *write*.

4.1.1.6 Périphériques et variables du noyau

Notre implémentation ne tient pas compte explicitement de ces objets. Linux assimile les périphériques à des fichiers. Par conséquent, les mêmes règles de traitement s'appliquent si un périphérique est accédé directement (par exemple, par une lecture ou écriture dans */dev/lp0*). En revanche, si un périphérique est accédé en tant que mémoire de masse, au travers d'un système de fichiers ou en tant que partition de *swap*, les flux d'informations sont contrôlés au niveau de l'accès aux fichiers ou aux pages mémoire. Enfin, les périphériques amovibles ne peuvent être gérés par notre implémentation, basée entièrement sur le contrôle des flux locaux au système.

Bien que les variables du noyau Linux soient des objets atomiques à part entière, aucune attaque à notre connaissance ne les exploite et nous ne contrôlons pas leur accès. Nous pouvons les assimiler à des canaux cachés.

4.1.2 Fichiers non-atomiques

Le modèle considère chaque fichier comme un objet atomique, dont le contenu peut être accédé grâce à la méthode *read* et modifié par la méthode *write*. En pratique, malheureusement, l'utilisation de certains fichiers ne correspond pas à cette vision : leur *contenu* n'est pas utilisé en tant qu'un tout indivisible.

4.1.2.1 Contenu non-atomique

Le *contenu* de certains fichiers dans Linux consiste en une agglomération d'informations indépendantes et accédées individuellement. Par exemple :
- le fichier */etc/shadow* contient les mots de passe des utilisateurs, à raison d'un mot de passe par ligne. Chaque mot de passe est indépendant des autres et peut être modifié seul ;
- une archive contient une section pour chaque fichier stocké, dont l'accès est indépendant. Les bibliothèques statiques (fichiers *lib***.a*) sont également des archives ;
- etc.

Dans le cadre de notre implémentation du modèle, tous les fichiers sont considérés atomiques. Il serait en effet très difficile de contrôler l'accès à des *sous-parties* de fichiers, notamment lorsque leur délimitation dépend du format du fichier ou de la syntaxe utilisée.

Cette simplification n'a de conséquences que dans les cas où les différentes sous-parties d'un fichier à contenu non-atomique doivent obéir à des règles d'accès différentes. Ces cas s'avèrent rares et concernent essentiellement les fichiers relatifs aux comptes et aux mots de passe des utilisateurs (*/etc/passwd*, */etc/shadow*, */etc/samba/smbpasswd*, etc.) Chaque utilisateur doit pouvoir, en effet, consulter ou modifier certains paramètres de son compte (mot de passe, *shell* de session, etc.) et uniquement du sien. Or, le fichier étant considéré atomique, une attaque qui modifie les paramètres du compte d'un *autre* utilisateur est potentiellement possible. Notre détecteur ne sera donc pas en mesure de détecter une telle attaque.

4.1.2.2 Solutions potentielles

Ce problème est connu [60] et peut être aisément résolu si un tel fichier est replacé par un *ensemble de fichiers* à contenu atomique, obéissant chacun à une politique de confidentialité/intégrité propre. Ceci peut être fait de deux manières :

1. en décomposant effectivement le fichier en un ensemble de fichiers, et en remplaçant les programmes qui utilisent ce fichier (par exemple, pour les fichiers */etc/passwd* et */etc/shadow*, il s'agit des commandes *chsh* et *passwd* ainsi que du module d'authentification) par des versions qui tiennent compte de cette décomposition ;
2. sans modifier ni remplacer le fichier, en créant une « projection » de son contenu sous forme d'un ensemble de fichiers indépendants, avec des politiques d'accès différentes. Le nouveau système de fichiers *Reiser4* [60], disponible dans la prochaine version du noyau Linux, offrira cette possibilité.

Nous avons choisi la seconde solution, car elle présente l'avantage d'assurer une compatibilité ascendante totale et ne nécessite pas de modification les logiciels existants. Notre détecteur pourra en tirer parti aussitôt qu'une version stable de ce système de fichiers sera diffusée.

On remarquera qu'à elle seule, l'implémentation d'une telle solution n'évacue pas le risque d'attaques : un utilisateur peut en effet toujours mettre en oeuvre une attaque par délégation pour lire ou modifier le contenu d'un fichier auquel il n'a pas accès selon la politique de sécurité. Par contre, les hypothèses du modèle seront alors remplies, ces attaques seront donc bien détectées.

4.1.3 Opérations

Dans Linux, les flux d'informations entre les objets du système ne peuvent être créés qu'au moyen des *appels systèmes*. Les opérations que nous devons considérer sont donc les appels systèmes créant des flux depuis ou vers les objets que nous prenons en compte, ainsi que les appels systèmes qui créent ces objets. Chacun de ces appels systèmes devra être modélisé sous forme de flux d'informations.

La table 4.2 liste les appels système qui produisent des flux entre objets. Pour certains de ces appels, les flux effectivement produits dépendent des paramètres fournis lors de l'exécution de l'appel :

- l'appel système *mmap* crée des flux entre un fichier et le segment de données du processus appelant ou son segment de « texte », selon la présence de l'option *MAP_EXEC* ;
- dans l'appel *shmat*, les pages « mappées » et l'espace d'adressage correspondant apparaissent en tant qu'objets source et/ou destination du flux, selon la présence des options *PROT_WRITE* et *PROT_READ* ;
- dans l'appel *open*, le fichier ouvert apparaît en tant qu'objet source ou destination, selon la présence des options *O_RDONLY*, *O_WRONLY*, *O_RDWR* et *O_APPEND*.

Les appels *open* et *creat* entraînent l'évaluation du nom du fichier passé en paramètre. L'objet *data* (c'est-à-dire, le segment de données du processus appelant) figure donc en tant qu'objet source.

Dans les autres cas, le flux modélise trivialement la sémantique de l'opération : par exemple, *sendfile* produit un flux depuis le descripteur source vers le descripteur destination.

4.2 Politiques de sécurité

Par défaut, les politiques de confidentialité et d'intégrité sont implémentées sous Linux au moyen du contrôle d'accès discrétionnaire (DAC). Nous pouvons en dériver un ensemble de domaines et une fonction d'état de la matrice de contrôle d'accès.

4.2.1 Droits d'accès

Sous Linux, une politique de contrôle d'accès s'exprime sous forme de droits de lecture (r) et d'écriture (w) attribués aux sujets (utilisateurs), soit au moyen des neuf bits de permissions standard d'Unix, soit sous forme d'ACL[1]. Les droits d'exécution (x) ne sont pas pris en compte dans notre cas, car ils ne correspondent pas à la possibilité de créer des flux d'informations entre objets. Lors du chargement d'un exécutable, celui-ci est mappé en mémoire et le flux correspondant est pris en compte par notre modèle de l'opération *mmap*.

Du point de vue des flux d'informations entre objets, nous considérons un flux comme *légal* s'il correspond à une opération autorisée pour un sujet par la politique. Un flux est *illégal* si aucun sujet n'est autorisé à exécuter une opération correspondante. Par conséquent, l'opération créant un flux d'informations

$$\{o_1.read...o_n.read\} \gg \{o'_1.write...o'_p.write\}$$

est légale si et seulement s'il existe un sujet u tel que :

1. u possède le droit r sur tous les objets o_1 à o_n dans la matrice DAC ;
2. pour chaque i dans $[1...p]$, o'_i est un objet créé par l'opération, ou u possède le droit w sur o'_i dans la matrice DAC.

Si l'on assimile le sujet u à un domaine, on représente respectivement ses droits r et w sur un objet o par des références $R_u o.read$ et $R_u o.write$ et la matrice DAC par une fonction d'état, alors ceci correspond à la règle d'unicité de domaine. Nous en déduisons l'algorithme 2, permettant de construire l'ensemble de domaines D et la fonction d'état initiale s_0. Son principe est le suivant :

- tous les flux autorisés *simultanément* (c'est-à-dire pouvant par exemple être créés par un même utilisateur au regard du contrôle d'accès) appartiennent à un même domaine ;

[1] Sous Linux, les ACL et les attributs librement définissables ne sont actuellement disponibles que sur les systèmes de fichiers JFS, XFS et les versions récentes de Ext3.

– un ensemble de flux qui ne sont *pas* autorisés simultanément (c'est-à-dire pouvant être créés par exemple par deux utilisateurs distincts) appartiennent à des domaines distincts.

On remarquera que si chaque domaine correspond à un sujet, tout sujet ne définit pas automatiquement un domaine. En effet, si l'ensemble des droits d'un sujet u est un sous-ensemble de celui d'un sujet u' et si un flux de données est autorisé par les droits de u, alors il l'est également par ceux de u'. Il suffit donc de considérer u' dans la vérification de la règle d'unicité de domaine.

Remarque 4.2.1 *Nous ne devons pas considérer le « super-utilisateur »* root *en tant que sujet. En effet, les droits de ce dernier sont illimités et n'entrent pas en compte dans la définition de la politique de confidentialité/intégrité. Toutefois, les attaques exploitant l'acquisition de l'identité* root *sont bien détectées, car elles créent précisément des flux d'informations qui ne sont autorisés pour aucun sujet considéré.*

Algorithm 2 Construction de D et de s_0:

- Données :
  1. U est l'ensemble des sujets ;
  2. O est l'ensemble des objets ;
  3. M est l'ensemble des méthodes.
- Initialement :
  1. $D = U$
  2. $\forall o \in O, s_0(o) = \emptyset$
- Pour chaque u dans U, pour chaque m dans M :
  1. Soit $objs_{u,m}$ l'ensemble des objets sur lesquels u est autorisé à invoquer la méthode m.
  2. S'il existe u' dans U, $u' \neq u$, tel que pour toute méthode m' dans M, $objs_{u,m'} \subseteq objs_{u',m'}$
     (a) alors $D \leftarrow D \setminus \{u\}$
     (b) sinon pour tout o dans $objs_{u,m}$, $s_0(o) \leftarrow s_0(o) \cup R_u o.m$

4.2.2 Objets virtuels et authentification

Par définition, les objets virtuels ne figurent pas dans la matrice de contrôle d'accès et les références les concernant ne seront donc pas définies automatiquement. Il existe trois cas de tels objets :

1. pour certains objets virtuels, tous les flux de données sont *a priori* autorisés car aucune restriction n'est imposée par la politique de sécurité : c'est le cas par exemple des objets virtuels qui représentent les sockets primaires de services réseau s'exécutant sous l'identité *root* (*sshd*, *lpd*, *webmin* etc.). Les ensembles de références associés à ces objets comprennent donc par défaut des références dans tous les domaines existants ;
2. certains objets virtuels sont créés par des opérations engendrant des flux à partir d'autres objets virtuels (par exemple, la création d'une socket secondaire lors de la connexion d'un client réseau). Le flux de références s'applique normalement dans ce cas ;
3. certains objets virtuels sont des objets terminaux (ce sont des sources ou des puits de données), représentant l'échange de données avec un utilisateur (console, socket de connexion distante, etc.). Ces objets sont créés par l'opération d'authentification de l'utilisateur et leur ensemble de références comprend les références de lecture et d'écriture dans le domaine associé (c'est-à-dire le domaine représentant cet utilisateur ou le domaine représentant un utilisateur dont les droits sont un sur-ensemble de ceux de cet utilisateur).

Dans le premier cas, la politique s'avère souvent trop permissive en pratique et il peut s'avérer utile de restreindre manuellement l'ensemble des flux d'informations possibles : on peut décider d'associer à ces objets virtuels des références appartenant à un ensemble de domaines plus réduit. Par exemple, on peut associer à la socket principale sur le port *TCP 80* (objet virtuel $tcp80$) l'ensemble de références :

$$\{R_{www}tcp80.read, R_{www}tcp80.write, R_{www}tcp80.accept\}$$

Ceci correspond au fait d'exécuter le service *HTTP* avec l'identité de l'utilisateur *www*, au lieu de celle de *root*.

Dans le troisième cas, une solution simple pour la prise en compte de l'authentification consiste à représenter chaque utilisateur u par un objet virtuel spécifique $useru$, avec l'ensemble de références

$$\{R_{du}useru.read, R_{du}useru.write\}$$

où du est le domaine associé à l'utilisateur u. L'authentification de u donne ainsi lieu à la création d'un ensemble d'objets virtuels terminaux $o_1...o_n$ et est modélisée sous forme d'un flux

$$\{useru.read\} \gg \{o_1.write...o_n.write\}$$

4.2.3 Identité des sujets

La règle d'unicité de domaine, appliquée à cette représentation de Linux, ne tient compte que des objets source et destination d'un flux d'information. La notion de « sujet exécutant une opération » disparaît ainsi totalement.

Remarquons avant tout que cette conséquence de l'utilisation de notre approche est cohérente avec l'objectif principal : détecter des violations de la politique de sécurité au moyen d'attaques par délégation. Le fait d'utiliser l'identité du sujet en tant que critère de légalité d'une opération constitue en effet la base sur laquelle reposent ces techniques d'attaque. Logiquement, la définition d'un modèle de détection de ces attaques nous conduit donc à éliminer la notion de « sujet » des critères de légalité.

Une opération sera donc considérée comme étant légale si elle satisfait la règle d'unicité de domaine, indépendemment de l'identité utilisateur associée au processus qui l'exécute. Toutefois, directement ou par compositions successives de flux causalement dépendants, l'opération peut aboutir à la création d'un flux depuis ou vers un objet virtuel représentant un utilisateur (cf. paragraphe 4.2.2). Si c'est le cas, la règle d'unicité de domaine assure effectivement la vérification des droits d'un utilisateur, où cet utilisateur n'est pas représenté par le processus exécutant l'opération, mais par la séquence causale d'opérations exécutées.

Cette situation est illustrée par exemple par l'attaque exploitant *lpr* (cf. paragraphe 3.1.2.1). Cette attaque aboutit par composition à la construction du flux

$$\{useralice.read, shadow.read\} \gg \{printer.write\}$$

où

- $useralice$ est l'objet virtuel représentant l'utilisateur *Alice* ;
- $shadow$ est le fichier */etc/shadow* ;
- $printer$ est l'objet représentant l'imprimante (par exemple, */dev/lp0*).

Si *Alice* n'est pas autorisée à lire */etc/shadow*, ce flux viole trivialement la règle d'unicité de domaine : l'envoi du document vers l'imprimante est par conséquent illégal.

4.2.4 Authentification

L'authentification d'un sujet est modélisée par un flux depuis l'objet virtuel représentant l'utilisateur vers les objets terminaux du processus authentifié : les descripteurs d'entrées/sorties standard (*stdin*, *stderr*, *stdout*) et les sockets.

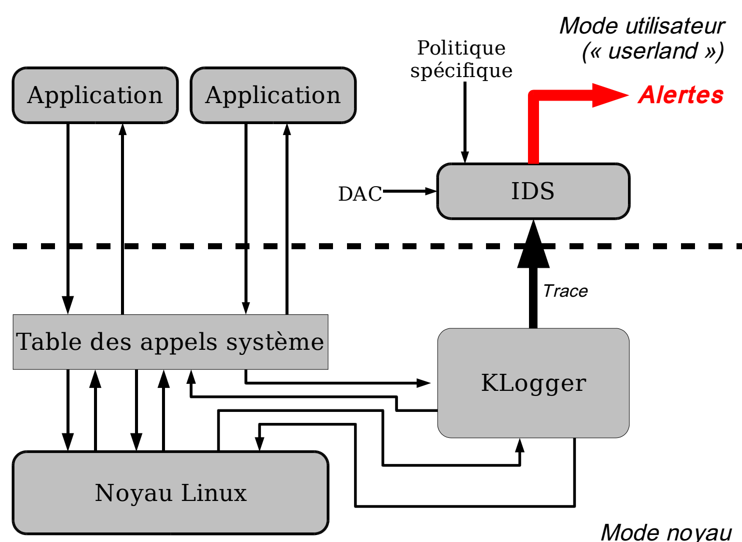


FIG. 4.1 – Redirection des appels système

4.3 Réalisation

En utilisant notre modèle, nous avons mis en oeuvre un détecteur d'intrusions lors de l'exécution[2]. Ce système est composé de trois modules principaux :

1. le générateur de traces (« *KLogger* ») ;
2. le générateur de l'état initial ;
3. le contrôleur de flux d'informations proprement dit.

Nous allons décrire brièvement chacun de ces trois modules.

4.3.1 Génération des traces

Afin de générer les traces des appels système, nous devons intercepter l'exécution de ces appels. Une solution simple consiste à détourner ces appels système et remplacer les fonctions correspondantes par nos propres versions, comme le résume la figure 4.1.

Concrètement, nous avons conçu un programme « *KLogger* », qui s'exécute en tant que module du noyau Linux. Lorsqu'il est activé, ce module remplace les entrées relatives aux appels système observés dans la table des appels système du noyau. Lorsqu'un de ces appels est invoqué, la fonction

[2] *Runtime IDS*

correspondante fournie par *KLogger* est exécutée. Chacune de ces fonctions procède en deux étapes :

1. la fonction originale qui traite effectivement l'appel système est invoquée ;
2. un descriptif de l'appel exécuté est ajouté à la trace, comprenant le type de l'appel système, la valeur des paramètres et celle du résultat renvoyé.

Du fait de l'ordonnanceur préemptif, il est possible que le processus appelant soit interrompu entre les étapes 1 et 2. L'ordre d'apparition des appels système dans la trace peut donc être différent de l'ordre dans lequel ils ont été effectivement exécutés. Ceci n'est pas un problème, dans la mesure où notre implémentation garantit deux propriétés :

1. l'ordre des appels système dans la trace respecte l'ordre local de leur exécution par chaque processus ;
2. l'ordre d'exécution des appels système relatifs à un même objet est préservé, grâce à une gestion du type *FIFO* par le module KLogger. Il n'est pas nécessairement identique à leur ordre d'invocation, mais la création des flux d'information dépend de l'ordre effectif d'exécution et non d'invocation.

Ces deux conditions sont suffisantes pour garantir que la trace respecte l'*ordre causal* des appels système. L'enchaînement des flux d'informations déduit de cette trace est donc conforme à l'exécution.

4.3.2 Génération de l'état initial

L'état initial est défini à partir de la politique de contrôle d'accès discrétionnaire en utilisant l'algorithme 2 de la page 79. Par ailleurs, les ensembles de références pour les objets virtuels peuvent être définis manuellement à l'aide d'un langage de déclaration. La figure 4.2 en présente un exemple : on déclare ici un ensemble de références *sampleweb*, comprenant les références *read* et *write* dans le domaine 10010 (chaque domaine possède un identificateur numérique, choisi arbitrairement). La syntaxe complète du langage de configuration est spécifiée en annexe (voir page 119).

```
# Sample domains for a web server

declare sampleweb { read 10010, write 10010, accept 10010 }

tcpport 80 = sampleweb
```

FIG. 4.2 – Exemple de déclaration d'objets virtuels

Le stockage des ensembles de références est persistant (les références concernant les fichiers sont stockées dans des attributs de ces derniers, les

références pour les objets virtuels sont stockées dans un répertoire dédié). L'état initial doit donc être généré une seule fois lors de l'installation du système de détection, une régénération est nécessaire en cas de modification de la politique, par exemple lors de la modification des droits d'accès par *chmod*.

Sur une station de travail Linux comprenant environ 100000 fichiers et 10 utilisateurs, dotée d'un disque dur IDE 7200 tours, le processus de génération complet dure environ 15 minutes.

4.3.3 Contrôle des flux

Le détecteur d'intrusions est un processus de basse priorité qui contrôle les flux d'informations à partir de l'état initial, en fonction des traces d'appels système produites par *KLogger*. Dans la version actuelle, *KLogger* communique avec le démon de détection par l'intermédiaire d'une file de messages. La détection se fait donc parallèlement à l'exécution mais de manière asynchrone. Du fait des temps de traitement et de la politique d'ordonnancement, notre implémentation ne garantit pas que l'alerte sera levée *avant* que l'opération qui viole la politique de sécurité n'ait été exécutée. Une telle garantie ne fait toutefois pas partie de nos objectifs dans le cadre de cette thèse.

Certaines informations nécessaires au contrôle des flux lors d'un appel système dépendent du processus exécutant cet appel, par exemple les objets liés aux descripteurs d'entrée/sortie ou les espaces d'adressage. En plus des appels système produisant effectivement ces flux que nous avons décrit dans la section 4.1.3, nous devons donc intercepter des appels permettant de maintenir ces informations à jour. Ce sont en particulier les appels système de création de processus et threads (*fork* et *clone*) où il est nécessaire de prendre en compte l' « héritage » des descripteurs par le processus fils, les appels de manipulation des descripteurs (*pipe*, *dup* et *close*) et les appels de manipulation des espaces d'adressage (*shm_open* et *shmctl*).

Nous avons enfin défini plusieurs messages de contrôle, pouvant être transmis au démon de détection afin de contrôler son fonctionnement (réinitialisation, gestion du fichier de « log », arrêt du détecteur).

4.4 Retour d'expérience

Cette réalisation a montré que notre approche est effectivement applicable à un système d'exploitation standard, moyennant quelques adaptations mineures telles que la gestion particulière des pages mémoire. Le système Linux suit très bien la représentation orientée objet que nous avons choisie, les exceptions (fichiers à contenu non-atomique, etc.) ne surviennent que dans des cas particuliers facilement identifiables.

La table 4.3 résume le coût en mémoire causée par le fonctionnement de notre détecteur sur un système typique. Dans ce système, chaque utilisateur (autre que *root*) donne lieu à un domaine ; nous avons défini par ailleurs un domaine d'accès à */etc/shadow* pour la modification des mots de masse, un domaine dédié pour le service HTTP (où seuls les transferts entre les fichiers composant le site Web et les sockets des clients sont autorisés), un domaine dédié aux connexions *ftp* anonymes et un domaine « vide », utilisé par défaut pour les autres connexions réseau entrantes.

Ce système est équipé de disques SCSI rapides, ce qui explique le temps de génération de l'état initial relativement court. La dégradation des performances du système due à notre détecteur est d'environ 15%, comme le montre la table 4.4.

Dans l'absolu, cette surcharge apparaît importante mais elle ne nous paraît pas prohibitive. Ce détecteur se montre donc effectivement utilisable en pratique. En outre, notre version expérimentale actuelle est conçue pour être facilement mise en oeuvre, sans aucun souci d'optimisation des performances. Une étude des possibilités d'amélioration des performances fera l'objet de nos futurs travaux.

Les tests de détection et les résultats obtenus sont présentés dans le chapitre 5.

4.5 Application à d'autres systèmes d'exploitation

L'étude d'une application de notre modèle sur un autre système d'exploitation que Linux ne rentre pas dans le cadre de cette thèse. Néanmoins, nous pouvons formuler quelques remarques sur ce sujet.

Notre modèle peut être en principe implémenté sur tout environnement qui remplit les conditions suivantes :

1. le système doit être compatible avec l'approche orientée objet ;
2. les opérations produisant les flux d'information entre objets doivent être clairement identifiées ;
3. il doit être possible d'intercepter ces opérations afin de produire les traces.

Nous avons vu que Linux satisfait les conditions 1 et 2. C'est également le cas de tout système basé sur l'architecture UNIX, où la quasi-totalité des éléments du système sont considérés comme des objets de type « fichier » et il n'est possible d'accéder aux objets qu'en utilisant un ensemble connu d'appels système.

La troisième condition est naturellement remplie dans le cas de Linux grâce à l'accès au code source du système et à la possibilité de le modifier. D'autres systèmes, notamment BSD, Solaris et Darwin, offrent cette facilité.

Les systèmes UNIX au code source non ouvert (AIX, HP-UX...) remplissent naturellement également les deux premières conditions. Ces systèmes offrent par ailleurs la possibilité d'intégrer au noyau des modules tiers, qui peut être utilisée, dans certains cas, pour intercepter les appels système. Il reste néanmoins nécessaire d'étudier la faisabilité technique réelle de cette solution.

Le système Microsoft Windows, dans ses versions NT 4.0, 2000 et XP, se montre très proche de l'architecture UNIX et l'approche orientée objet s'applique naturellement. Grâce au contrôle d'accès discrétionnaire au niveau objet utilisée par Windows, l'algorithme 2 de génération de l'état initial est utilisable. On ne rencontre pas le problème des fichiers à contenu non-atomique [61] : dans le cas de Windows, chaque information élémentaire est stockée soit dans un fichier dédié, sont dans la « base des registres » dont les entrées sont par définition atomiques. Sans avoir accès au code source du système, il est néanmoins possible d'intercepter les appels système *Win32*. Par conséquent, il est possible d'implémenter notre détecteur sur ce système.

appel système	représentation
open(file,O_RDONLY)	$\{file.read, data.read\} \gg \{data.write\}$ *data* est le segment de données du processus appelant
fd=open(file,O_WRONLY)	$\{data.read\} \gg \{file.write, fd.write\}$
fd=open(file,O_RDWR) *ou* fd=open(file,O_WRONLYO_APPEND)	$\{file.read, data.read\} \gg$ $\{file.write, data.write\}$
fd=creat(file)	$\{data.read\} \gg \{file.write, fd.write\}$
read(fd,buf...)	$\{obj.read\} \gg \{data.write\}$ *obj* est l'objet ouvert sous *fd* sockets : $\{obj.read\} \gg \{data.write, obj.write\}$
write(fd,buf...)	$\{data.read\} \gg \{obj.write\}$
sendfile(out,in)	$\{in.read\} \gg \{out.write\}$
sec=accept(prim)	$\{prim.accept\} \gg \{sec.write\}$
send(sock,msg)	$\{data.read\} \gg \{sock.write\}$
recv(sock,msg)	$\{sock.read\} \gg \{data.write\}$
msgsnd(msgid,msg)	$\{data.read\} \gg \{msgid.write\}$
msgrcv(msgid,msg)	$\{msgid.read\} \gg \{data.write\}$
shmat(shmid,PROT_READ)	$\{shmid.map_r\} \gg \{data.write\}$
shmat(shmid,PROT_WRITE)	$\{data.read\} \gg \{shmid.map_w\}$
shmat(shmid,PROT_READ+PROT_WRITE)	$\{shmid.map_r, data.read\} \gg$ $\{shmid.map_w, data.write\}$
mmap(MAP_EXEC,...,fd)	$\{file.read\} \gg \{exec.write\}$ *file* est le fichier mappé *exec* est le segment texte du processus appelant
mmap(!MAP_EXEC,...,fd)	$\{file.read\} \gg \{data.write\}$
msync(...) *ou* munmap(...)	$\{data.read\} \gg \{file.write\}$
setxattr(file,attr,...)	$\{data.read\} \gg \{file : attr.write\}$ *file :attr* est l'attribut *attr* du fichier *file*
getxattr(file,attr,...)	$\{data.read, file : attr.read\} \gg \{data.write\}$

TAB. 4.2 – Appels système pris en compte

Nombre de fichiers	121965
Nombre d'utilisateurs	17
Nombre total de domaines	21
Stockage des références	94 Mo
Temps de génération de l'état initial	11 min 34 s
Occupation de mémoire vive	7,2 Mo

TAB. 4.3 – Surcharge mémoire causée par le détecteur

nombre d'opérations	12051434
système standard	9m 40s
avec détecteur actif	11m 23s
surcharge	15%

Table 4.4: Compilation complète d'un noyau Linux

Chapitre 5

Utilisation et tests

Nous avons mené deux séries d'expériences, respectivement en environnement contrôlé et en environnement d'exploitation réelle. Pour chacune de ces séries, nous décrivons successivement notre plate-forme de tests, les politiques de sécurité et les attaques qui ont été exécutées. Nous présentons alors les résultats respectifs obtenus et nous discutons de leur interprétation.

Les tests et les résultats décrits dans ce chapitre ont fait l'objet d'une publication [62].

5.1 Tests en environnement contrôlé

L'objectif de ces tests est de valider notre modèle de détection dans un contexte parfaitement maîtrisé. L'occurrence d'attaques étant connue, nous pouvons directement interpréter la présence ou l'absence d'alertes dans la sortie du détecteur.

5.1.1 Environnement de tests

Nous avons implémenté notre détecteur d'intrusions sur un système Debian GNU/Linux 3.0 avec un noyau 2.4.18, sur processeur Intel Celeron. Nous utilisons un environnement complet de station de travail, incluant le serveur Web *BuggyHTTP* [63]. Ce serveur offre un service *HTTP* minimal, avec un grand nombre de vulnérabilités. Nous l'avons choisi pour la facilité avec laquelle ces vulnérabilités peuvent être exploitées. Néanmoins, il offre un prototype relativement fidèle des attaques potentielles contre les produits tels qu'*Apache* ou *Microsoft IIS*. Nous utilisons exclusivement des attaques contre ce serveur.

Pendant toute la durée des tests, notre ordinateur est relié à un réseau local, relié à aucun environnement extérieur. De la sorte, nous pouvons nous assurer qu'aucune attaque inconnue ou inattendue ne se produira au cours des tests. Aussi, nos résultats refléteront directement la capacité de notre

Fichier ou Répertoire	Propriétaire	Groupe	Permissions
/var/www/index.html	*www-data*	*www-data*	`-rw-r--r--`
/etc/shadow	*root*	*shadow*	`-rw-r-----`
/tmp	*root*	*root*	`drwxrwxrwt`
/home/alice/doca.txt	*alice*	*alice*	`-rw-r--r--`
/home/bob/docb.txt	*bob*	*bob*	`-rw-r--r--`

TAB. 5.1 – Fichiers utilisés lors des attaques

détecteur à signaler les attaques que nous avons mises en oeuvre.

5.1.2 Politiques de sécurité

Nous utilisons successivement deux politiques de sécurité. Le serveur *BuggyHTTP* fonctionne toujours avec les droits de *root*. Les domaines ont été générés automatiquement en fonction des droits d'accès résumés dans la table 5.1, en utilisant l'algorithme 2. Seuls les bits de permissions classiques ont été utilisés (sans ACL étendues). *BuggyHTTP* ne procède à aucune authentification, par conséquent, aucun utilisateur ne possède de droits particuliers sur l'objet virtuel « *socket HTTP* ».

Les groupes d'utilisateurs sont constitués ainsi :

- le seul membre de *www-data* est l'utilisateur *www-data* ;
- le groupe *shadow* est vide ;
- le seul membre de *root* est *root* ;
- les seuls membres des groupes *alice* et *bob* sont respectivement *alice* et *bob*.

Dans un premier temps, nous n'avons pas apporté de précision à cette politique de sécurité. En particulier, l'objet *socket sur le port 80* (le port 80 étant utilisé pour la socket du service *BuggyHTTP*) n'est soumis par défaut à aucun contrôle d'accès, si bien que tous les flux entrants et sortants (autres que ceux pouvant uniquement être créés par *root*) sont autorisés. Le graphe des flux autorisés entre les objets dont nous tenons compte dans ces tests est présenté à la figure 5.1.

On remarquera que cette politique autorise tous les flux depuis les objets « publiquement lisibles » vers la socket du serveur.

Dans une seconde phase, nous avons restreint davantage la politique, afin de n'autoriser vers la socket du service *HTTP* que les flux provenant des fichiers faisant réellement partie du site Web, en l'occurrence *index.html*. Pour cela, nous avons introduit un nouveau domaine (portant arbitrairement le numéro 10020) et nous avons déclaré, dans la configuration du détecteur, les références suivantes :

```
# Restriction de la politique de contrôle d'accès
```

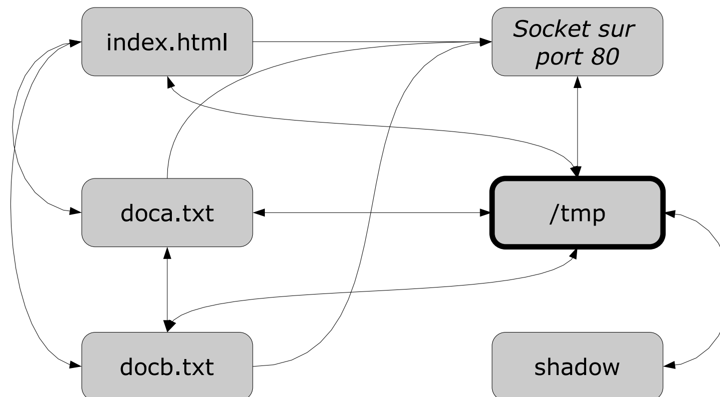


FIG. 5.1 – Flux autorisés par le contrôle d'accès

```
tcpport 80 = { read 10020, write 10020 }
file '/var/www/index.html' = { read 10020 }
```

Les flux autorisés dans ce cas sont présentés par la figure 5.2.

5.1.3 Description des attaques

Nous avons exploité différentes vulnérabilités du serveur *BuggyHTTP*, de manière à produire les effets suivants :

1. Accès à des fichiers, soit faisant partie du site (par exemple *index.html*), soit des fichiers confidentiels (par exemple */etc/shadow*) ;
2. Modification de fichiers ;
3. Exécution d'opérations locales sur le serveur.

Nous détaillons ci-après les vulnérabilités exploitées pour obtenir ces effets.

5.1.3.1 URL illégales

Le serveur *BuggyHTTP* n'effectue aucun contrôle des chemins d'accès dans les URL. Il est donc possible de l'utiliser pour accéder au contenu de fichiers lisibles uniquement par *root*, par exemple au moyen de la requête :

```
GET /../../etc/shadow HTTP/1.1
```

Le détecteur produit une alerte, qui correspond au flux illégal de */etc/shadow* vers la socket client.

En revanche, la requête suivante ne donne pas lieu à une alerte :

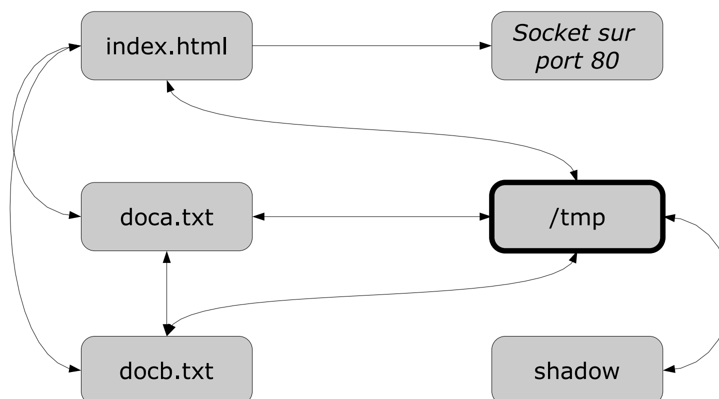


FIG. 5.2 – Flux autorisés par la politique affinée

```
GET /../../var/www/index.html HTTP/1.1
```

Bien qu'utilisant le même principe, cette requête produit un flux légal (*index.html* vers la socket client).

On peut remarquer que les systèmes actuels se protègent contre ce type d'attaque par une réécriture d'URL, ou par des signatures qui détectent la présence du motif *../..* dans les URL. Dans notre cas, le détecteur distingue la violation effective de la politique d'une utilisation bénigne de la vulnérabilité.

5.1.3.2 Exécution de programmes *cgi-bin*

Il est possible d'exécuter un programme arbitraire grâce à des requêtes *cgi-bin*. Nous pouvons par exemple obtenir un interpréteur de commandes au moyen de la requête :

```
GET /cgi-bin/../../bin/sh
```

Cet interpréteur étant exécuté par le serveur *BuggyHTTP*, il en hérite l'identité de l'utilisateur *root* et, par conséquent, les droits. De fait, nous obtenons un accès en tant que *root*. Aucune alerte n'est produite à ce stade : en effet, il n'y a aucune violation de politique de confidentialité ni d'intégrité. Par contre, nous pouvons utiliser ce *shell* dans différents buts :

– accéder au fichier *index.html* par une commande telle que :

  ```
  cat /var/www/index.html
  ```

  Aucune alerte ne se produit, car nous accédons de fait à un fichier du site Web par l'intermédiaire du serveur Web (par connexion sur le port TCP 80), bien que de manière détournée ;

- accéder à un fichier confidentiel, par une commande telle que par exemple :

  ```
  cat /etc/shadow
  ```

  Ceci est une violation effective de la politique de confidentialité et on observe bien une alerte ;
- modifier un fichier, l'exemple le plus simple étant une commande telle que :

  ```
  cat > /var/www/index.html
  ```

  Ceci est une violation effective de la politique d'intégrité (aucun flux depuis un client du service *HTTP* n'est autorisé vers les fichiers locaux sur le serveur). Nous observons bien une alerte ;
- exécuter une opération locale sur le serveur. Nous traiterons de ce cas dans la section 5.1.4.

5.1.3.3 Débordements de tampon

Le serveur *BuggyHTTP* ne contrôle pas la taille des données transmises dans les requêtes ; une attaque par débordement de tampon (*buffer-overflow*) est par conséquent possible. Cette faille peut être exploitée soit à des fins de déni de service, soit pour laisser le serveur exécuter une opération particulière.

Notre approche n'est pas adaptée à la détection des dénis de service, aussi, une telle attaque reste indétectée. Par contre, nous avons exploité cette faille en utilisant un « shellcode » pour obtenir un accès en *root*, que nous pouvons mettre à profit de manière similaire à l'accès obtenu *via* cgi-bin.

5.1.3.4 Attaques par fenêtre temporelle (*Race-conditions*)

Nous avons modifié le serveur *BuggyHTTP* afin d'introduire une faille de sécurité supplémentaire du type *race-conditions*. Par défaut, chaque cycle du serveur consiste en deux phases :

1. le serveur enregistre la requête reçue dans un fichier dont le nom est de la forme

   $$/var/www/log/ip.num$$

   où *ip* désigne l'adresse IP du hôte serveur et *num* le numéro de série de la requête ;
2. le serveur renvoie une réponse au client.

Remarque 5.1.1 *Les fichiers de* log *sont créés par des flux provenant de la socket utilisée par le client. Un flux utilisant ces fichiers en tant que source n'est donc autorisé que si un flux vers la/les même(s) destination(s) serait possible en utilisant comme source la socket client.*

Nous avons inversé l'ordre des deux étapes, ce qui ouvre la voie à des attaques du type *race-conditions*. Si le client est en mesure de prévoir le nom du fichier où sa requête sera consignée (c'est-à-dire s'il peut deviner le numéro de série de sa prochaine requête), il peut par exemple créer un lien symbolique portant ce nom et pointant sur un fichier inaccessible *via* le service Web. Ceci aura pour effet d'écraser le contenu de ce dernier fichier et, par conséquent de violer la politique d'intégrité.

Nous avons utilisé concrètement l'attaque suivante :

1. le client consulte le répertoire des fichiers de *log* et regarde le numéro de série de la dernière requête traitée. Le fichier où sa propre requête sera consignée portera alors ce numéro augmenté de 1 ;
2. le client crée un lien portant ce nom et pointant sur */etc/shadow* ;
3. il fait exécuter une opération quelconque (par exemple, lire *index.html*) ;
4. le contenu de */etc/shadow* est écrasé lorsque le serveur consigne cette requête.

Il existe différentes manières d'effectuer les étapes 1 et 2, le plus simple consiste à obtenir un *shell* en tant que *root*, au moyen d'une attaque par *cgi-bin* ou *buffer-overflow*. Dans tous les cas, le flux créé à l'étape 4 est un flux de la socket client vers */etc/shadow* et donne lieu à une alerte.

5.1.4 Opérations locales exécutées

Nous avons testé l'exécution des opérations locales suivantes, à l'aide d'un shell *root* obtenu par une attaque utilisant *cgi-bin* :

1. création d'un nouveau fichier :

   ```
   cat > /tmp/myfile
   Hello World
   ^D
   ```

   Cette opération ne viole pas la confidentialité ni l'intégrité d'une information résidente. Nous n'obtenons pas d'alerte, car la création d'un nouvel objet par un flux d'informations est toujours possible.

2. copie d'un fichier créé vers un fichier existant :

   ```
   cp /tmp/myfile /var/www/index.html
   ```

   Nous modifions ici un fichier existant par un flux composé dont la source originale est la socket *HTTP*. Un tel flux est interdit par la politique de sécurité et nous obtenons une alerte dans ce cas.

3. copie locale de fichiers, créant un flux interdit :

   ```
   cp /home/alice/doca.txt /etc/shadow
   ```

Ce flux est explicitement interdit par la politique et nous obtenons une alerte.

4. copie locale de fichiers, créant un nouvel objet :

```
cp /home/alice/doca.txt /tmp/copie
```

De même que lors de la création directe d'un nouveau fichier lors du test n° 1, le flux n'est pas illégal. Par ailleurs, le modèle utilisé de l'appel système *open* nécessite des références de lecture pour le fichier source (*doca.txt*) et pour l'objet mémoire contenant le nom du fichier. Ces références existent dans un même domaine avec la politique de sécurité par défaut et aucune alerte n'est produit. Ce n'est pas le cas de la seconde politique affinée, où un tel domaine n'existe pas. Une alerte est alors levée. Selon l'interprétation, nous pouvons considérer le premier cas (pas d'alerte) comme un faux négatif, ou le second cas (alerte produite) comme un faux positif car la politique n'est pas explicitement violée. Dans les deux cas, une copie ultérieure de */tmp/copie* vers un fichier existant crée un flux explicitement illégal, donnant lieu à une alerte.

5. copie locale de fichiers, créant un flux légal :

```
cp /home/alice/doca.txt /home/bob/docb.txt
```

En soi, ceci ne viole pas la politique (un flux depuis *doca.txt* vers *docb.txt* est autorisé). Néanmoins, un client Web ne doit pas être en mesure de déclencher une telle opération. Le modèle de l'appel système *open* que nous avons défini dans la section 4.1.3 du chapitre 4 tient compte des flux d'informations concernant les paramètres de cet appel. Comme les flux depuis un client Web vers *docb.txt* ne sont pas autorisés, nous obtenons une alerte dans ce cas.

5.1.5 Tests et résultats

La table 5.2 présente les résultats des tests où les quatre techniques d'attaque décrites précédemment sont utilisées pour accéder aux fichiers composant le site Web (colonne *site*), accéder à des fichiers ne faisant partie du site mais dont l'accès est autorisé à quiconque par le contrôle d'accès discrétionnaire (colonne *public*), accéder à des fichiers dont l'accès n'est autorisé qu'à des utilisateurs privilégiés (colonne *confidentiel*), et enfin pour modifier un fichier local, ce qu'un client Web n'est pas autorisé à faire (colonne *modification*). La colonne *log* correspond au cas particulier de l'accès (en lecture ou écriture) aux fichiers de journal du serveur *BuggyHTTP*.

Chaque case du tableau indique dans quel(s) cas le détecteur a produit une alerte :

- une case vide indique qu'aucune alerte n'a été produite ;

attaque	site	public	confidentiel	modification	log
URL		2	1,2	N/A	
cgi-bin		2	1,2	1,2	
buffer-overflow		2	1,2	1,2	
race-condition	N/A	N/A	N/A	1,2	

TAB. 5.2 – Tests avec *BuggyHTTP*

- 1,2 indique qu'une alerte a été émise avec chacune des deux politiques de sécurité applicables ;
- 2 indique qu'une alerte n'a été émise que lorsque la seconde politique de sécurité a été appliquée.

Ces tests ont été répétés plusieurs fois dans un ordre différent. Les résultats sont constants et reproductibles.

On observe en premier lieu qu'aucune alerte n'a été émise lors d'un accès uniquement au site Web, quelle que soit la technique utilisée (*URL* standard ou non, attaque par *cgi-bin* ou par *buffer-overflow*). Aucun « faux-positif » n'a donc été produit au cours de ces tests. Seules les violations effectives des propriétés de la politique sont détectées.

L'accès à des fichiers lisibles par tous les utilisateurs (par exemple *doca.txt*) n'a été détecté que lorsque la politique de sécurité restreinte a été appliquée. Ce résultat est conforme aux attentes : dans la politique de sécurité par défaut, l'accès à ces fichiers est autorisé et le contrôle d'accès discrétionnaire ne distingue pas un accès local d'un accès à distance par l'intermédiaire du service *HTTP*. Si le détecteur agit correctement, le contrôle d'accès discrétionnaire seul apparaît comme insuffisant pour spécifier la politique de sécurité d'un tel service. En revanche, préciser suffisamment la politique s'avère extrêmement simple avec notre prototype (voir section 5.1.2).

L'accès à des fichiers confidentiels (en l'occurrence, */etc/shadow*) a été détecté correctement dans tous les cas, de même que les tentatives de modification de fichiers. Ces dernières sont détectées quel que soit le fichier visé (*index.html*, */home/alice/doca.txt* ou */etc/shadow*). La politique de sécurité interdit dans tous les cas toute modification de fichier par un client Web, c'est à dire par un utilisateur non authentifié (aucun fichier n'est modifiable par « tout le monde »).

Les fichiers de *log* sont créés par des flux dont la source est la socket du service *HTTP*. La politique de sécurité n'introduit aucune précision concernant ces flux. D'après le modèle, tous les flux sont donc autorisés entre la socket *HTTP* et ces fichiers, ainsi qu'entre ces fichiers. Aucune alerte n'a donc été produite dans les cas suivants :

1. accès au contenu d'un fichier de *log* existant ;
2. modification d'un fichier de *log* existant ;

3. copie d'un fichier de *log* existant vers un autre fichier de *log*.

Il convient d'observer qu'aucune de ces trois actions ne viole la politique de contrôle d'accès strictement parlant. Néanmoins, nous pouvons raisonnablement considérer la modification d'un fichier de *log* existant comme une atteinte à l'intégrité du système. De même, l'accès à un fichier de *log* existant sera probablement interprété comme une violation de la confidentialité du système. Par défaut, notre approche n'est pas en mesure de détecter ces attaques, car le système n'autorise pas de distinction entre les opérations exécutées lors de deux connexions successives au service *HTTP*. Le problème ne se poserait pas si *BuggyHTTP* utilisait le service système *syslog*. Dans ce cas, le journal apparaîtrait comme un objet atomique, les flux depuis la socket *HTTP* vers le journal étant autorisés.

Nous observons enfin que dans tous les cas, les résultats sont identiques quelle que soit la technique d'attaque utilisée, et dépendent uniquement de l'effet produit par l'attaque. Par ailleurs, pour les deux politiques de sécurité utilisées, toutes les attaques qui violent effectivement la politique sont détectées. Ceci correspond à toutes les attaques qui modifient des fichiers ou accèdent à des fichiers non publics dans le premier cas et à toutes les attaques testées dans le second cas).

5.1.6 Discussion

Au cours de ces tests, les résultats obtenus ont été conformes aux attentes, de manière reproductible. Par ailleurs, toutes les alertes correspondent effectivement à des violations effectives de la politique.

L'accès ou la modification des seuls fichiers de *log* n'est pas détectée. Ceci est dû au fait qu'en l'état, notre modèle du système ne permet pas de prendre en compte le fait que chaque fichier de *log* est créé par une connexion indépendante. L'utilisation du service de journalisation *syslog* permet de résoudre le problème.

La création de fichiers locaux n'est en principe pas considérée comme une violation de la politique et ne donne pas lieu à des alertes. Néanmoins, certains utilisateurs peuvent la juger indésirable car elle peut conduire par exemple à un déni de service, par saturation de l'espace disque ou de la mémoire vive. Les dénis de service n'entrent toutefois pas dans le cadre de notre approche.

Mis à part ces limites, la détection d'une violation des propriétés de la politique est indépendante de l'attaque utilisée et une même attaque peut donner lieu à une alerte ou non, selon qu'elle aboutit effectivement à une violation explicite de la politique ou pas. Ces résultats sont donc encourageants. Nous pouvons conclure que notre détecteur fonctionne conformément aux attentes et que l'approche est validée dans le cadre de ces exemples d'attaques typiques.

	Système 1	Système 2
Architecture	Pentium III 450 MHz	Pentium III 450 MHz
Utilisation	serveur *WWW* et mail	station de travail
Système d'exploitation	Debian Linux 3.0	RedHat Linux 6.2
Version du noyau	2.4.18-xfs	2.2.16

TAB. 5.3 – Systèmes testés

5.2 Tests en utilisation réelle

Nous avons testé le détecteur d'intrusions dans des conditions d'exploitation réelle, où nous avons également exécuté des attaques exploitant des failles des logiciels installés. Notre objectif consiste ici à évaluer l'efficacité pratique de notre approche de détection d'intrusions.

5.2.1 Environnement de tests

Nous utilisons deux systèmes de tests, dont les principales caractéristiques sont résumés dans la table 5.3. Le système 1 est utilisé par l'équipe *SSIR* du campus de Rennes de Supélec en tant que serveur de courrier électronique et serveur Web. Le système 2 est une station de travail où nous avons délibérément installé une version de Linux possédant de nombreuses failles de sécurité, en l'occurrence RedHat 6.2.

Lors des tests, les logiciels suivants sont en service sur le système 1 :

Serveur Web Apache version 1.3.36. Ce serveur peut être configuré pour stocker son journal dans un fichier dédié ou utiliser le service standard *syslog*. D'après notre expérience avec *BuggyHTTP*, nous avons choisi cette seconde option ;

Serveur SMTP Exim version 3.35 ;

Serveurs POPS et IMAPS uw-imap version 2003-Debian ;

SSH2 version 2.0.13

Sur le système 2, nous utilisons *exim* version 3.13 et *OpenSSH* 2.9p2, utilisant l'option *UseLogin*. Ce système étant plus vulnérable, nous l'utilisons pour mener à bien plusieurs attaques volontaires et vérifier si elles sont signalées par le détecteur. Certaines de ces attaques ne fonctionnent pas contre le système 1, aussi, nous pouvons vérifier l'absence d'alertes dans ce cas. Enfin, le système 1 nous permet d'évaluer le nombre de « *faux-positifs* » au cours d'un fonctionnement typique.

Les deux systèmes sont reliés à un réseau local, utilisé par l'équipe *SSIR* (15 enseignants-chercheurs et doctorants). Le détecteur d'intrusions à base de signatures *snort*, utilisé sur ce réseau, produit entre 100 et 1000 alertes par jour. Compte tenu de la situation particulière de ce réseau (protection

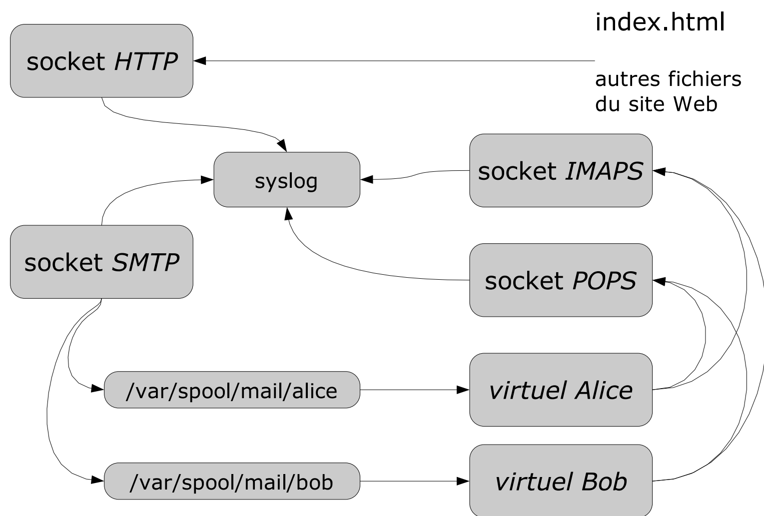


FIG. 5.3 – Politique de sécurité définie manuellement sur les systèmes 1 et 2

par deux *firewalls* vis-à-vis de l'extérieur, utilisateurs internes dignes de confiance) et de l'expérience passée, nous pouvons considérer que virtuellement 100% de ces alertes sont des faux positifs.

5.2.2 Politique de sécurité

Nous avons soumis les deux systèmes à la même politique de sécurité. En plus des domaines déduits automatiquement du contrôle d'accès, nous avons défini manuellement une politique concernant le Web et les services de courrier électronique. Cette politique est résumée par le graphe de la figure 5.3.

5.2.2.1 Service Web

En ce qui concerne le service Web, la politique que nous avons mise en oeuvre est similaire à celle décrite dans la section 5.1.2. Seuls les fichiers faisant partie du site Web doivent être accessibles par une connexion sur la socket du service *HTTP*. Par ailleurs, aucun objet existant ne peut être modifié par un flux ayant pour source cette socket. La seule exception est l'objet virtuel représentant *syslog*, à destination duquel tous les flux sont par défaut autorisés.

Nous avons donc créé un nouveau domaine (portant l'identificateur arbitraire 10020), en attribuant à la socket sur le port 80 les références *read* et *write* dans le domaine 10020. Nous avons associé à chaque fichier du site Web une référence *read* dans le domaine 10020. Cette procédure a été automatisée grâce à un script qui extrait automatiquement les noms de tous les fichiers

référencés à partir du point d'entrée du site *index.html*. La configuration résultante est :

```
# Web-related security policy
declare website { read 10020 }
declare web_io { read 10020, write 10020 }
tcpport 80 = web_io
syslog add web_io
file '/var/www/index.html' = website
file '/var/www/search.html' = website
etc...
```

5.2.2.2 Services de courrier électronique

Nous n'avons défini aucune politique en ce qui concerne le courrier électronique sortant. La politique concernant la réception et la consultation des boîtes aux lettres est la suivante :

- un client connecté sur la socket du service *SMTP* peut écrire du courrier électronique dans chaque boîte aux lettres, mais dans aucun autre fichier ;
- les boîtes aux lettres ne peuvent être consultées qu'en se connectant aux sockets des services *POPS* ou *IMAPS*, avec authentification.

Cette politique ne peut pas être automatiquement dérivée du contrôle d'accès. Comme celui-ci ne distingue pas par défaut un accès local d'un accès distant, il autoriserait soit les flux depuis la socket du client *SMTP* vers *tous* les objets d'un utilisateur donné, soit vers *aucun* objet de l'utilisateur. De même, la politique d'utilisation de ce serveur implique que les utilisateurs ne doivent pas consulter leur courriel au moyen d'opérations locales, mais uniquement par l'intermédiaire du service *POPS* ou *IMAPS*. Le contrôle d'accès standard ne permet pas d'exprimer cette contrainte.

Nous définissions donc pour chaque utilisateur un nouveau domaine dédié au courrier électronique. L'identificateur de ce domaine est arbitrairement $20000 + user_id$ de l'utilisateur. Les références sont associés aux objets de la façon suivante :

- chaque boîte aux lettres possède des références *read* et *write* dans le « domaine de courriel » de son propriétaire ;
- la socket du service *SMTP* possède une référence *read* dans le « domaine de courriel » de chaque utilisateur ;
- la socket du services *SMTP* possède des références *read* et *write* dans un domaine (d'identificateur arbitraire 50000) n'autorisant aucun flux, à l'exception des flux vers *syslog*. Ces références sont utilisées pour dialoguer avec le client ;
- les sockets des services *POPS* et *IMAPS* ne possèdent par défaut que des références dans un domaine (d'identificateur arbitraire 60000) n'autorisant aucun flux, à l'exception des flux vers *syslog*. Ces références

sont utilisées pour dialoguer avec le client ;
- l'objet virtuel de chaque utilisateur possède, en plus des références par défaut, les références *read* et *write* de son « domaine de courriel ». Conformément à notre modèle de Linux, celles-ci sont propagées vers les sockets des services *POPS* et *IMAPS* si les serveurs respectifs exécutent une authentification appropriée ;
- dans chaque domaine de courriel, les flux vers *syslog* sont autorisés.

La construction de cette configuration a été également automatisée à l'aide d'un script. Les déclarations résultantes sont :

```
# E-mail related policy
# SMTP
declare smtprefs { read 50000, write 50000 }
tcpport 25 = smtprefs
syslog add smtprefs
# POP & IMAP
declare mailaccessrefs { read 60000, write 60000 }
tcpport 993 = mailaccessrefs
tcpport 995 = mailaccessrefs
syslog add mailaccessrefs
# alice (uid = 1007)
tcpport 25 add { read 21007 }
declare alicemail { read 21007, write 21007 }
file '/var/spool/mail/alice' = alicemail
virtual 1007 add alicemail
syslog add alicemail
# bob (uid = 1004)
etc...
```

5.2.3 Description des attaques

Nous avons exploité plusieurs vulnérabilités connues. Celles-ci ont été choisies aléatoirement, à condition qu'au moins un des deux systèmes sous test soit vulnérable. Nous imitons ainsi le comportement d'un attaquant « type », qui essaiera d'utiliser les outils d'attaque dont il dispose, en fonction des connaissances qu'il a pu obtenir de notre/nos système(s).

Nous présentons ci-après les vulnérabilités que nous avons exploitées, puis les attaques que nous avons exécutées.

5.2.3.1 Erreur de format dans Exim (CVE-2001-0690)

Cette faille affecte la version 3.13 d'Exim, proposée dans la distribution RedHat 6.2 et utilisée sur notre système n° 2. Ce système est vulnérable si nous activons l'option *headers_check_syntax* d'Exim. Le système n° 1 n'est pas vulnérable.

Lorsque la chaîne de caractères contenant l'adresse de retour d'un courriel (le contenu du champ *From :*) contient des caractères de formatage (par exemple, *%n* ou *%s*), un effet de bord dans le code source d'Exim entraîne leur remplacement en fonction du contenu des registres du processeur et de la pile. Cette chaîne étant elle-même stockée sur la pile, nous pouvons dès lors provoquer un débordement de tampon ayant pour effet d'écraser le sommet de la pile, contenant entre autres, la valeur sauvegardée du compteur ordinal. De la sorte, il est possible d'exécuter un shell *root*.

5.2.3.2 Erreur de format dans rpc.statd (CVE-2000-0666)

Cette faille est similaire à la précédente. Le serveur *rpc.statd*, utilisé pour verrouiller les fichiers exportés *via* le protocole *NFS*, est sujet à une attaque par erreur de format dans la chaîne de caractères identifiant le hôte. Lorsque cette chaîne est transmise au service de journalisation *syslog*, son interprétation peut entraîner la modification de valeurs sur la pile, dont celle du compteur ordinal. Il est possible ainsi d'exécuter un shell *root*.

Le système n° 2 est vulnérable à cette attaque.

5.2.3.3 *Race-condition* dans le noyau Linux (CAN-2003-0127)

Cette faille, communément appelée « ptrace-kmod », affecte tous les noyaux Linux de version inférieure à 2.4.21. Nos deux systèmes sont par conséquent vulnérables.

Linux utilise un modèle de threads particulier, dans lequel une tâche partageant l'espace d'adressage de son « père » peut invoquer les appels système de la famille *exec* de manière à exécuter un programme différent. Néanmoins, la fonctionnalité *ptrace* reste disponible dans ce cas. Grâce à elle, la tâche père peut contrôler pas à pas l'exécution du « fils ».

Ce fonctionnement de *ptrace* est une erreur de conception qui ouvre la voie à une attaque par *race-condition* :

1. l'attaquant provoque le chargement d'un module du noyau. Ceci donne lieu à la création d'un thread « fils » privilégié par le système *kmod* de Linux ;
2. l'attaquant prend le contrôle de ce fils par *ptrace* ;
3. l'exécution du fils est suspendue ;
4. l'attaquant peut alors remplacer le code exécutable du fils par sa propre version, qui démarre généralement un shell ;
5. l'exécution du fils reprend et le shell est exécuté ;
6. l'attaquant obtient un shell *root*.

5.2.3.4 Autres

Nous avons également utilisé les attaques contre *Apache* et *OpenSSH* déjà présentées dans les sections 3.1.2.3 et 3.1.2.2. Ces attaques permettent respectivement d'exécuter à distance un shell avec les droits d'accès de *www-data* et d'obtenir un shell *root* local à partir d'une simple session utilisateur. Le système n° 2 est vulnérable à ces deux attaques.

5.2.4 Tests et résultats

Nous avons utilisé les failles décrites précédemment pour tenter de violer la politique de confidentialité et d'intégrité sur nos deux systèmes de tests.

5.2.4.1 Utilisation des « exploits »

Nous avons réalisé les violations de politique de confidentialité et d'intégrité suivantes :

1. accès distant à des fichiers confidentiels ou modification illégale de fichiers à distance, en utilisant CVE-2001-0690, CVE-2000-0666 et la combinaison de la vulnérabilité [56] + CAN-2003-0127 ;
2. accès ou modification en tant qu'utilisateur local de fichiers en violant les droits d'accès, par CAN-2003-0127 ou la vulnérabilité [55] ;
3. remplacement du service *HTTP* par un serveur de shell *root*. Pour cela, nous avons utilisé la vulnérabilité [56] puis CAN-2003-0127 de manière à obtenir un accès *root* au hôte cible, puis nous avons arrêté Apache et l'avons remplacé par un interpréteur de commandes attaché à la socket *HTTP*. Toute connexion à cette socket donne désormais immédiatement accès à un shell *root* ;
4. écrasement ou modification arbitraire des boîtes aux lettres des utilisateurs, en utilisant CVE-2001-0690.

Les mêmes procédures d'attaque ont été utilisées contre nos deux systèmes de tests, pendant une période de tests de 10 jours. En plus des attaques (improbables, mais théoriquement possibles) dont nous n'avons pas eu connaissance durant cette période, nous avons exécuté 120 fois les scénarios d'attaques, dont certains sans violer effectivement la politique de sécurité. Compte tenu des vulnérabilités respectives des deux systèmes, nous avons ainsi entraîné 50 violations effectives.

Compte tenu du très grand nombre d'opérations traitées (plus de 26 millions), les violations effectives sont submergées par du bruit. Dans la suite de cette section, nous détaillons les résultats de ces tests résumés dans la table 5.4.

Nombre total d'événements	26276342
Occurrences des scénarios d'attaque	120
Nos violations effectives	50
Détections correctes	38
Autres alertes	6
Faux positifs	4
Faux négatifs	12

TAB. 5.4 – Résumé des résultats des tests

5.2.4.2 Détections correctes

Parmi les 50 violations effectives que nous avons effectuées, 38 ont directement donné lieu à des alertes. Toutes correspondent soit à un accès en lecture à un fichier confidentiel, soit à une modification illégale d'un fichier. Ces violations couvrent l'ensemble des méthodes que nous avons utilisées, soit :

- exécution directe d'une opération distante ;
- obtention directe d'un shell *root* local ou distant ;
- création puis utilisation d'un « serveur de shell *root* ».

L'utilisation de scénarios identiques à l'encontre d'un système non-vulnérable, ou l'utilisation de ces scénarios sans violer effectivement de politique (par exemple, en exploitant [56] pour accéder à *index.html*) n'a *pas entraîné d'alerte.*

Le détecteur a par ailleurs levé 6 autres alertes, qui ne correspondent pas à nos propres attaques. Ces alertes correspondent aux cas suivants :

1. utilisation par un utilisateur d'un logiciel de courrier électronique local sur le système n° 1, en accédant directement à la boîte aux lettres dans */var/spool/mail* ;
2. accès par un utilisateur à des fichiers dans son répertoire personnel (accessible par *NFS* sur le système n° 1) *via* le service *IMAPS* ;
3. création d'une sauvegarde, copiant l'ensemble des fichiers dans une destination unique. Le périphérique correspondant étant vu du détecteur comme un fichier ordinaire, cette sauvegarde est interprétée comme un ensemble de flux convergeants vers une destination unique, ce qui viole les domaines définis par la politique de sécurité.

Les deux premiers cas correspondent à des violations de notre politique de sécurité : en effet, celle-ci impose aux utilisateurs de consulter leur courrier électronique en utilisant les services *POPS* ou *IMAPS* et interdit explicitement la consultation locale sur le serveur. De même, la politique interdit l'accès aux données autres que les boîtes aux lettres *via* ces services.

Le troisième cas (lecture simultanée de tous les fichiers) n'est pas pris en compte dans la politique et constitue, de fait, une anomalie au regard de

celle-ci. Cette alerte ne correspond donc pas directement à une attaque, mais signale néanmoins une mise en défaut de la politique de sécurité.

De façon générale, nous pouvons dire que ces alertes sont donc dues à une politique de sécurité trop stricte.

5.2.4.3 Faux négatifs

Parmi les 50 intrusions effectives, 12 n'ont pas été détectées. Elles correspondent aux cas suivants :

1. accès distant à un fichier lisible par tous les utilisateurs en utilisant une attaque exploitant CVE-2000-0666 ;
2. modification d'une boîte aux lettres, en y stockant uniquement des données provenant du réseau ;
3. lecture du contenu d'une boîte aux lettres *via* le service *SMTP*, en exploitant CVE-2001-0690.

Le premier cas est dû à une spécification insuffisante de la politique. En effet, le contrôle d'accès ne distingue pas un accès local d'un accès distant. D'autre part, nous n'avons pas défini manuellement de politique concernant le service *rpc.statd*, donc les domaines générés autorisent les flux depuis les fichiers publiquement lisibles vers les sockets des clients de *rpc.statd*.

Le second cas est dû au fait que le contenu d'une boîte aux lettres n'est pas atomique. Le détecteur ne peut pas distinguer un flux ajoutant un nouveau message d'un flux modifiant un message stocké précédemment dans la boîte aux lettres. Ce problème serait résolu si chaque message était stocké dans un fichier indépendant, mais cela exigerait une modification du format de stockage des courriels et, par conséquent, des serveurs *SMTP*, *POP/POPS* et *IMAP/IMAPS*

Le troisième cas traduit une limite de notre approche. Le contenu d'une boîte aux lettres étant considéré comme une copie du contenu de la socket du client *SMTP*, un flux de la boîte aux lettres vers cette dernière est par défaut autorisé. Ce problème serait résolu si nous utilisions une authentification *SMTP*, ou si nous modifiions le modèle afin que les flux créés lors de chaque connexion d'un client appartiennent à un domaine distinct, de manière à différencier les flux créés par plusieurs connexions différentes.

5.2.4.4 Faux positifs

Au cours de ces tests, le détecteur a également produit 4 alertes supplémentaires que nous devons considérer comme des faux positifs (c'est à dire des alertes provoquées par des comportements *par définition légaux*).

L'analyse a montré que trois de ces alertes correspondent à des connexions à distance sur le système n° 1 *via* SSH2, en utilisant l'authentification par clef publique. Cette méthode d'authentification n'utilise pas le système

PAM et n'est donc pas prise en compte comme telle par notre modèle de Linux. L'accès à la base des clefs publiques et l'ouverture de la session de l'utilisateur connecté est dont interprété comme une violation de la politique de confidentialité. Ces alertes sont donc dues à une adéquation insuffisante entre le système réel et sa modélisation.

Les causes de la quatrième fausse alerte ne sont pas déterminées à l'heure actuelle.

5.2.5 Discussion

Au cours de ces tests, toutes les attaques violant sans ambiguïté la politique de sécurité ont été détectées, à l'exception des accès illégaux aux boîtes aux lettres *via SMTP* qui sortent du cadre du modèle à l'heure actuelle. Les alertes générées correspondent à des *violations effectives* de la politique, et non à des *occurrences des attaques* : l'occurrence d'une attaque qui ne viole pas la politique n'est pas signalée.

Le taux de faux positifs reste bas : seulement 4 fausses alertes ont été observées alors que plus de 26 millions d'événements ont été traités. De plus, au moins 3 de ces alertes correspondent à un fonctionnement débordant du cadre du modèle (authentification utilisant un mécanisme non modélisé).

Ces tests mettent par ailleurs en évidence la nécessité de définir une politique de sécurité et un modèle du système suffisamment précis. Une simple prise en compte de la politique de contrôle d'accès locale s'avère insuffisante dès lors que des services réseau sont utilisés. En effet, on souhaite généralement que l'ensemble des données accessibles par l'intermédiaire de ces derniers soit beaucoup plus restreint que l'ensemble des données accessibles aux utilisateurs locaux. Le modèle conduit par défaut à considérer certains comportements comme légaux, alors qu'ils violent de fait la politique de sécurité (dans notre cas, l'exploitation d'une faille dans le serveur *SMTP* pour accéder aux boîtes aux lettres). Ceci correspond aux limites d'une approche orientée système : ces flux d'informations ne sont pas illégaux sur le strict plan de la politique de sécurité *locale*, en revanche, ils se montrent illégaux dès lors que nous prenons en compte les liaisons externes du système. Par exemple, le modèle ne tient pas compte du fait qu'une socket n'est pas un conteneur d'informations au même titre qu'un fichier mais qu'elle crée des flux d'informations non contrôlables sur le système local. Il est donc nécessaire de raffiner davantage la politique, comme nous l'avons fait pour le courrier électronique. Une solution pratique serait d'utiliser des listes de contrôle d'accès étendues, en définissant un utilisateur particulier « distant ».

L'adéquation du modèle du système au système réel se montre également cruciale. Une fonctionnalité du système non modélisée (dans notre cas, l'authentification par clé publique dans *SSH2*) peut donner lieu à des faux positifs.

Néanmoins, en tenant compte de ces conditions et limites, notre approche montre ici sa capacité à détecter des intrusions de manière à la fois fiable et pertinente, en se basant uniquement sur un modèle du système et de la politique de sécurité.

Chapitre 6

Conclusion

Dans cette thèse, nous avons présenté une approche de détection d'intrusions paramétrée par la politique basée sur le contrôle des flux d'informations entre les objets système. Cette approche peut être utilisée pour détecter des violations de confidentialité et d'intégrité en utilisant uniquement un modèle du système d'exploitation cible et de sa politique de sécurité.

Nous avons volontairement écarté les solutions nécessitant une phase d'apprentissage ou d'expertise du comportement sain attendu ou des scénarios d'attaque. En choisissant une approche basée exclusivement sur la politique de sécurité, sans la paramétrer aucunement par le comportement usuel des utilisateurs, des applications ou des attaquants, nous sommes à même de détecter les violations *effectives* de la dite politique, quand bien même cette violation serait due à une attaque inconnue et/ou nouvelle. De notre point de vue, cette propriété est une qualité importante pour un détecteur d'intrusions.

Cette approche repose sur une définition formelle des violations de politique. Le système est modélisé sous forme d'un ensemble d'objets et de flux d'informations possibles entre ces objets. Ces flux sont produits par un ensemble d'opérations exécutées lors du fonctionnement du système, chaque opération exécutée se traduisant par un flux d'informations. La politique de sécurité consiste alors en une spécification des flux légaux. Une opération qui crée un flux considéré illégal, soit directement, soit par un enchaînement de flux causalement liés, viole par définition la politique.

Le principal outil utilisé pour mettre en oeuvre une politique de confidentialité et d'intégrité dans les systèmes actuels est le contrôle d'accès discrétionnaire. Nous avons interprété les droits d'accès spécifiés en termes de flux d'informations légaux. De la sorte, notre approche de détection complète le système de contrôle d'accès de manière transparente pour l'utilisateur. Nous avons cependant constaté qu'en pratique, une telle politique se montre insuffisamment précise, en particulier lorsqu'il s'agit de services accessibles à distance au travers du réseau. Dans ces cas, il est généralement souhaitable

de raffiner davantage cette politique, principalement en déclarant comme interdits certains flux pourtant légaux au regard du contrôle d'accès.

Nos tests ont montré l'efficacité de notre approche et sa capacité à détecter des attaques variées de manière fiable, tout en produisant un nombre très réduit de fausses alertes. L'utilisation de notre détecteur est fonctionnellement transparente pour le système et ne nécessite aucune modification des logiciels exploités, ni des droits existants des utilisateurs. Parallèlement à ces résultats encourageants, les tests nous ont permis de souligner l'importance de modéliser le système d'exploitation cible de manière suffisamment précise et la nécessité d'élaborer une politique de sécurité permettant de discriminer les flux d'informations de manière non ambiguë.

Nous pouvons espérer améliorer la pertinence et la fiabilité de la détection d'intrusions apportées par notre approche en suivant quatre axes de travail : améliorer la qualité de l'implémentation, améliorer en amont la spécification de la politique de sécurité, améliorer en aval le traitement et l'analyse des violations de cette politique et, à plus long terme, envisager l'utilisation du modèle en prévention des intrusions par contrôle actif de la politique.

Amélioration de l'implémentation logicielle

Nous devons améliorer l'adéquation du modèle utilisé au système réel. Il est possible d'étendre le modèle afin de prendre en compte certains cas particuliers non traités actuellement, comme nous l'avons vu sur l'exemple de l'authentification par clef publique avec *SSH2*. Inversement, nous pouvons faire évoluer le système lui-même afin d'en faciliter la modélisation. A titre d'exemple, le principal problème actuellement est que certains fichiers sont non-atomiques de fait. L'utilisation du nouveau système de fichiers *Reiser4*, qui permettra de les remplacer par un ensemble de fichiers atomiques de manière transparente pour les applications, semble prometteuse.

Enfin, il nous paraît indispensable d'étendre l'implémentation à un environnement distribué tel qu'un réseau local avec partage de fichiers et de services entre hôtes. La définition des domaines devra alors tenir compte des flux d'informations entre hôtes. Le contrôle des flux de références inter-hôtes qui en résulte devra s'appuyer sur un mécanisme d'authentification des références basé par exemple sur une signature numérique. Si le modèle reste inchangé, son implémentation se rapproche alors du concept de « capacités cachées » [64].

Traitement en amont : spécification des politiques de sécurité

La définition d'une politique de sécurité est un problème à part entière. Nous avons vu que la politique de confidentialité et d'intégrité définie par le contrôle d'accès discrétionnaire, tel qu'il est disponible par défaut sous

Linux, s'avère dans quelques cas insuffisamment précise. Dans notre implémentation, nous utilisons un langage de configuration permettant d'affiner la politique si nécessaire, sans toutefois prétendre à offrir un véritable langage de définition de politique.

De nombreux travaux ont été effectués dans le domaine de la spécification et la validation des politiques de sécurité. Une approche particulièrement intéressante consiste en une intégration avec DTE, en raison de la grande expressivité de ce dernier et de sa conception adaptée à l'utilisation dans un environnement général (Linux). Concrètement, cette démarche implique de pouvoir interpréter une politique spécifiée en DTEL sous forme de flux d'informations, comme nous l'avons fait dans le cas du contrôle d'accès discrétionnaire. L'architecture de DTE, orientée principalement vers l'isolation de programmes privilégiés, est bien adaptée à une telle démarche.

D'autre part, un détecteur d'intrusions paramétré par la politique nécessite une gestion efficace de la politique. En particulier, il est important que l'administrateur dispose d'une vision globale de celle-ci. Il est nécessaire de développer des outils d'administration adaptés, permettant de travailler à différents niveaux de granularité et exploitant éventuellement la représentation graphique des domaines de flux d'informations, avec une possibilité d'édition interactive. L'objectif de cette démarche doit être de faciliter autant que possible la mise en oeuvre d'une politique de sécurité donnée.

Traitement en aval : analyse des violations de la politique

A l'heure actuelle, notre version du détecteur d'intrusions ne produit que des alertes. Or, une utilisation pratique nécessiterait davantage d'informations, en particulier une analyse *a posteriori* de la séquence d'opérations menant à la violation de politique signalée. En effet, seule une connaissance du scénario de l'attaque permettrait à l'administrateur du système de mettre en oeuvre des mesures préventives, afin d'éviter la reproduction de l'attaque dans l'avenir.

Théoriquement, notre approche se prête bien à l'implémentation d'une telle fonctionnalité. Une solution simple consisterait à associer, lors de chaque flux, les identificateurs des objets source à chaque références propagée. En cas d'occurrence d'un flux illégal, il suffirait alors d'examiner la liste des identificateurs d'objets associés aux références utilisées pour reconstituer la séquence des flux aboutissant au flux illégal. Une telle solution exigerait cependant la gestion efficace d'un grand volume de données supplémentaires. Nos travaux futurs immédiats porteront sur l'étude d'une telle approche.

Vers un contrôle actif de la politique ?

Étendre notre approche en vue de prévenir ou de tolérer les intrusions fournit une piste de recherche particulièrement intéressante. En utilisant un

mécanisme de synchronisation simple, il est possible d'adapter notre implémentation afin de garantir qu'une alerte sera levée *avant* l'exécution effective de l'opération fautive. Néanmoins, une solution réellement efficace pour la prévention et la tolérance aux intrusions consisterait, soit à bloquer les opérations violant la politique, soit à les isoler de manière à en contrôler les effets. Ces approches sont séduisantes, mais leur faisabilité en utilisant notre modèle reste un problème ouvert. L'utilisation d'une infrastructure transactionnelle au niveau du système de fichier est une piste de recherche possible.

Bibliographie

[1] John McHugh. Intrusion and intrusion detection. *International Journal of Information Security*, July 2001.

[2] Yves Deswarte. *Sécurité des Réseaux et Systèmes Répartis*, chapter La sécurité des systèmes d'information et de communication, pages 15–65. 2003.

[3] J.P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Company, Fort Washington, Pennsylvania, April 1980.

[4] Sandeep Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, 1995.

[5] J. Allen, A. Christie, W. Fithen, J. McHugh, J. Pickel, and E. Stoner. State of the practice of intrusion detection technologies. Technical Report SEI-99TR-028, CMU/SEI, 2000.

[6] Cédric Michel and Ludovic Mé. ADeLe : an attack description language for knowledge-based intrusion detection. In *Proceedings of the 16th International Conference on Information Security (IFIP/SEC 2001)*, pages 353–365, June 2001.

[7] S. Eckmann, G. Vigna, and R. Kemmerer. Statl : An attack language for state-based intrusion detection. http ://citeseer.nj.nec.com/eckmann00statl.html, 2000.

[8] Frédéric Cuppens and Rodolphe Ortalo. Lambda : A language to model a database for detection of attacks. In H. Debar, L. Mé, and S. F. Wu, editors, *Proceedings of the Third International Workshop on the Recent Advances in Intrusion Detection (RAID'2000)*, number 1907 in LNCS, pages 197–216, October 2000.

[9] Ludovic Mé. Gassata, a genetic algorithm as an alternative tool for security audit trails analysis. Web proceedings of the First international workshop on the Recent Advances in Intrusion Detection (RAID'98), http ://www.raid-symposium.org/raid98, September 1998.

[10] William DuMouchel. Computer intrusion detection based on Bayes factors for comparing command transition probabilities. Technical Report TR91, National Institute of Statistical Sciences (NISS), February 1999.

[11] A. Mounji and B. Le Charlier. Detecting breaches in computer security : A pragmatic system with a logic programming flavor, 1996.

[12] Terran D. Lane. *Machine Learning Techniques for the computer security domain of anomaly detection.* PhD thesis, Department of Electrical and Computer Engineering, Purdue University, August 2000.

[13] R. Sekar, Y. Cai, and M. Segal. A specification-based approach for building survivable systems. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 338–347, 1998.

[14] R.K. Cunningham, R.P. Lippmann, and S.E. Webster. Detecting and displaying novel computer attacks with macroscope. *IEEE Transactions on Systems, Man & Cybernetics, Part A (Systems & Humans)*, 31(4) :275 – 81, July 2001.

[15] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3) :151–180, 1998.

[16] Benjamin Morin, Ludovic Mé, Hervé Debar, and title = Mireille Ducassé.

[17] Calvin Ko, George Fink, and Karl Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, Orlando, FL, 1994. IEEE Computer Society Press.

[18] R. Sekar, T. Bowen, and M. Segal. On preventing intrusions by process behavior monitoring. In *Proceedings of the 1999 USENIX Intrusion Detection Workshop*, pages 29–40, 1999.

[19] Calvin Ko, Manfred Ruschitzka, and Karl Levitt. Execution monitoring of security-critical programs in distributed systems : A specification-based approach. In *IEEE Symposium on Security and Privacy*, pages 175–187, 1997.

[20] Prem Uppuluri and R. Sekar. Experiences with specification-based intrusion detection. In W. Lee, L. Mé, and A. Wespi, editors, *Proceedings of the Fourth International Symposium on the Recent Advances in Intrusion Detection (RAID'2001)*, number 2212 in LNCS, pages 172–189, October 2001.

[21] Calvin Ko and Timothy Redmond. Noninterference and intrusion detection. In *Proccedings of the IEEE Symposium on Security and Privacy*, 2002.

[22] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 75–86, April 1984.

[23] Fred B. Schneider. Enforceable security policies. *Information and System Security*, 3(1) :30–50, 2000.

[24] M.A. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in operating systems. *ACM*, 19(8) :461–471, August 1976.

[25] Vijay Manian. Access leak problem in access control matrices. http ://www.cise.ufl.edu/ srkamath/access_control.ps, June 2002.

[26] R. S. Sandhu. The typed access matrix model. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122–136, 1992.

[27] M. Dacier and Y. Deswarte. Privilege graph : an extension to the typed access matrix. In *Proceedings of the European Symposium on Research in Computer Security ESORICS*, 1994.

[28] Rodolphe Ortalo, Yves Deswarte, and Mohamed Kaaniche. Experimenting with quantitative evaluation tools for monitoring operational security. *IEEE Transactions of Software Engineering*, 25(5) :633–650, 1999.

[29] D. Bell and L. LaPadula. Secure computer systems : Unified exposition and multics interpretation. Technical report, The Mitre Corp., 1976.

[30] J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994.

[31] K. Biba. Integrity considerations for secure computer systems. *MTR-3153, Mitre Corporation*, 1975.

[32] David F.C. Brewer and Michael J. Nash. The chinese wall security policy. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 206–214. IEEE Computer Society Press, May 1989.

[33] R. S. Sandhu. A lattice interpretation of the chinese wall policy. In *Proc. 15th NIST-NCSC National Computer Security Conference*, pages 329–339, 1992.

[34] A.K. Jones, R.J. Lipton, and L. Snyder. A linear time algorithm for deciding security. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, 1976.

[35] M. Bishop. Conspiracy and information flow in the take-grant protection model. *Journal of Computer Security*, 4(4) :331–359, 1996.

[36] M. Bishop and L. Snyder. The transfer of information and authority in a protection system. In *Proceedings of the Seventh Symposium in Operating Systems Principles*, pages 45–54, December 1979.

[37] D. Denning. A lattice model of secure information flow. In *Communications of the ACM 19 :5*, pages 236–243, 1976.

[38] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7) :504–513, 1977.

[39] J. Fenton. Information protection systems. *The Computer Journal*, pages 143–147, 1974.

[40] D. Volpano and C. Irvine. Secure flow typing. *Computer and Security*, 1997.

[41] P. Wadler. The marriage of effects and monads. In *International Conference on Functional Programming*, September 1998.

[42] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.

[43] John McLean. Security models and information flow. In *IEEE Symposium on Security and Privacy*, pages 180–189, 1990.

[44] Thomas P. Jensen, Daniel Le Metayer, and Tommy Thorn. Verification of control flow based security properties. In *IEEE Symposium on Security and Privacy*, pages 89–103, 1999.

[45] Andrew C. Myers. JFlow : Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[46] E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 130–140, 1997.

[47] Li Gong. Secure java class loading. *IEEE Internet Computing*, 2(6) :56–61, 1998.

[48] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Shermann, and Karen A. Oostendorp. Confining root programs with domain and type enforcement (DTE). In *Proceedings of the Sixth USENIX Security Symposium*, 1996.

[49] S. Cheung. An intrusion tolerance approach for protecting network infrastructures. *Ph.D. Dissertation, University of California, Davis*, September 1999.

[50] Yves Deswarte, L. Blain, and Jean-Charles Fabre. Intrusion tolerance in distributed computing systems. In *IEEE Symposium on Security and Privacy*, pages 110–121, 1991.

[51] J. Zimmermann, L. Mé, and C. Bidan. Introducing reference flow control for detecting intrusion at the os level. In *Proceedings of the Fifth International Symposium on Recent Advances in Intrusion Detection*, pages 292–306, October 2002.

[52] J. Zimmermann, L. Mé, and C. Bidan. An improved reference flow control model for policy-based intrusion detection. In *Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS)*, 2003.

[53] The MITRE CVE vulnerability list. http ://www.cve.mitre.org/cve/.

[54] CMU CERT/CC. Ca-1995-02 : Vulnerabilities in /bin/mail. http ://www.cert.org/advisories/CA-1995-02.html, January 26 1995.

[55] CMU CERT/CC. Vu#40327 : Openssh uselogin option allows remote execution of commands as root. http ://www.kb.cert.org/vuls/id/40327, November 2001.

[56] CVE. Can-2003-0084, 2003.

[57] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7) :558–565, July 1978.

[58] The usermode linux kernel project. http ://user-mode-linux.sourceforge.net/.

[59] mi2g Intelligence Unit. Linux remains most attacked server OS, September 2003.

[60] The reiser4 filesystem. http ://www.namesys.com/v4/v4.html.

[61] Michael M. Swift, Peter Brundrett, Cliff Van Dyke, Praerit Garg, Anne Hopkins, Shannon Chan, Mario Goertzel, and Gregory Jensenworth. Improving the granularity of access control in windows nt. In *Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 87–96. ACM Press, 2001.

[62] J. Zimmermann, L. Mé, and C. Bidan. Experimenting with a policy-based hids based on an information flow control model. In *Proceedings of the 19 Annual Computer Security Applications Conference (ACSAC)*, 2003.

[63] Giovanni Vigna. The buggy web server (tm). http ://www.cs.ucsb.edu/ vigna/courses/CS279/HW5/buggy.c.

[64] Daniel Hagimont, Jacques Mossiere, Xavier Rousset de Pina, and F. Saunier. Hidden software capabilities. In *International Conference on Distributed Computing Systems*, pages 282–289, 1996.

Annexe A

Langage de configuration

Syntaxe

Le langage de configuration obéit à la grammaire suivante :

```
config : ligne*

ligne : commentaire | déclaration | affectation

commentaire : '#' caractère* '\n'

déclaration : 'declare' ident liste_références

affectation : objet mode ident
   | objet mode liste_références

ident : caractère_alphanumérique+

liste_références : '{' listref '}'
listref : référence ',' listref | référence
référence : méthode entier
méthode : 'read' | 'write' | 'accept'

mode : '=' | 'add'

objet : 'file' '\'' chemin_de_fichier '\''
   | 'tcpport' entier
   | 'udpport' entier
   | 'virtual' entier
```

La notation $a*$ signifie « séquence de zéro ou plusieurs éléments a ». La notation $a+$ signifie « séquence d'un ou plusieurs éléments a ».

Objets

Les fichiers pouvant être déclarés dans le langage de configuration sont :

1. les sockets sur port *TCP*, identifiées par leur numéro de port, par exemple

   ```
   tcpport 80
   ```

2. les sockets sur port *UDP*, identifiées par leur numéro de port, par exemple

   ```
   udpport 120
   ```

3. les fichiers, identifiées par leur chemin d'accès, par exemple

   ```
   file '/tmp/helloworld'
   ```

4. les objets virtuels représentant les utilisateurs, identifiés par le numéro *user_id* de l'utilisateur, par exemple

   ```
   virtual 1002
   ```

Affectation des références aux objets

Il est possible de spécifier soit directement une liste de références, par exemple :

```
tcpport 80 = { read 10010, write 10010, accept 10010 }
```

soit un ensemble de références préalablement défini par une clause *declare* :

```
declare web { read 10010, write 10010, accept 10010 }
tcpport 80 = web
```

L'affectation des références aux objets peut se faire selon deux modes :

1. L'affectation directe, notée =, spécifie explicitement l'ensemble de références pour un objet. Par exemple :

   ```
   tcpport 80 = { read 10010, write 10010, accept 10010 }
   ```

   signifie que l'objet « socket sur port *TCP* 80 » possède les trois références citées ;

2. L'ajout, noté *add*, permet d'ajouter un ensemble de références à celles déjà définies pour l'objet. Par exemple :

   ```
   tcpport 80 = { read 10010, write 10010, accept 10010 }
   ```

signifie que les trois références citées sont ajoutées aux références existantes pour l'objet *tcpport 80*.

La génération automatique précède toujours la lecture du fichier de configuration, aussi, on considère dans la configuration que les références générées automatiquement existent.

Annexe B

Flux de références et non-interférence

L'approche à base de non-interférence proposée dans [21], que nous appellerons ici « *KR* »[1], a pour but de détecter les violations des politiques du type « le groupe de sujets G interfère avec la donnée d ». Nous montrons dans cette section que ces politiques peuvent être représentées dans notre modèle et, par conséquent, l'approche que nous proposons permet également d'en détecter les violations.

B.1 Sujets et données

Dans *KR*, une politique de sécurité est définie en termes d'un groupe de sujets G et d'une donnée d. Plus précisément, d représente une variable dont la valeur est indépendante des opérations exécutées par G. De plus, d'après les hypothèses, d est accessible au travers des opérations du système (par exemple, les appels système). Dans notre cas, d correspond donc naturellement à un *objet atomique*.

La notion de « sujet » , et donc de « groupe de sujets » n'existe pas dans le cadre de notre modèle, où chaque opération est pleinement déterminée uniquement par l'ensemble des objets source et destination. Contrairement à *KR*, l'opération n'est pas liée à un « processus exécutant » associé à une identité de sujet. Il n'y a donc pas de représentation directe dans notre modèle d'un « groupe de sujets » au sens de *KR*.

Nous représenterons un utilisateur par un *ensemble d'objets terminaux*, correspondant à son interface utilisateur. Dans un système du type UNIX, par exemple, l'interface utilisateur d'un sujet consiste en :

– un ensemble de terminaux virtuels (*/dev/tty**) ;
– une connexion au serveur d'affichage X11 (socket Unix ou TCP) ;

[1] D'après ses auteurs *Ko* et *Redmond*

– éventuellement d'autres objets particuliers.

L'action d'un sujet se traduit par un flux d'informations utilisant au moins un de ses objets terminaux en tant qu'objet source. Nous pouvons donc définir la notion de « sujet » par la dépendance causale : une opération est exécutée par un sujet si elle est une conséquence causale d'une action sur l'interface utilisateur de ce dernier. Plus précisément :

Propriété B.1.1 *Une opération est exécutée pour le compte d'un sujet* u *si elle crée par composition un flux d'informations utilisant au moins un objet terminal de* u *en tant qu'objet source.*

Dans certains cas, un sujet ne possède pas d'interface utilisateur associée (par exemple, le sujet *nobody* dans un système Linux). La définition n'en est pas moins valable, car tout sujet possède au moins un objet terminal : des données présentes en mémoire qui seront utilisées par les opérations exécutées (par exemple, la zone *data* d'un programme exécuté).

Remarquons que la règle énoncée ci-dessus est plus stricte que celle implicitement utilisée par *KR* :

Propriété B.1.2 *Dans* KR*, une opération est exécutée pour le compte d'un sujet* u *si elle fait partie d'un processus possédant l'identité* u*.*

Par exemple, si le sujet *Alice* crée un fichier dont le contenu sera ensuite lu par le sujet *Bob*, cette dernière opération apparaîtra, dans notre cas, à la fois comme une opération exécutée pour le compte d'*Alice* et pour le compte de *Bob*. Cependant, il découle trivialement de la règle B.1.1 que toute opération considérée comme étant exécutée pour le compte de u dans *KR* le sera également dans notre cas.

B.2 Politique de non-interférence

La notion de non-interférence utilisée dans *KR* se traduit, dans notre cas, par la déclaration de certains flux d'informations illégaux.

Étant donnée une donnée d dans la politique *KR*, nous notons $M_w(d)$ l'ensemble des méthodes modifiant l'état de l'objet atomique d, c'est à dire utilisables pour créer des flux d'informations où d figure en tant qu'objet de destination. Par ailleurs, étant donné un sujet u dans la politique *KR*, notons $term(u)$ l'ensemble des objets terminaux associés à u dans notre modèle.

La détection des violations de la politique *KR* s'appuie sur le principe suivant :

Théorème B.2.1 *Étant donnés un ensemble de sujets* KR G*, une donnée* KR d*, un système* (O, M, Ω) *et une politique de sécurité* (D, s) *tels que* $d \in O$

et $\forall u \in G, term(u) \subset O$*, toute violation de la politique* KR *« G n'interfère pas avec d » entraîne une violation de la politique* (D, s) *si :*

$$\forall u \in G, \forall o \in term(u), \forall m \in M, \forall m' \in M_w(d), dom(o.m, s) \cap dom(d.m', s) = \emptyset. \ (*)$$

Preuve :

Appelons KR_p la politique *KR* « G n'interfère pas avec d ». Considérons une trace :

$$[src_1 \gg dst_1, src_2 \gg dst_2, \ldots, src_n \gg dst_n]$$

1. Si $d \notin dst_i$ pour tout $i \leq n$, alors il n'y a pas de violation de politique KR_p car la valeur de d n'est pas modifiée.
2. Si la politique KR_p est violée, alors la valeur de d est modifiée. Dans ce cas, il existe i tel que $d \in dst_i$. S'il existe $m \in M$ et $tu \in term(u)$, u étant un utilisateur du groupe G, tels que $tu.m \in src_i$, alors $(*)$ implique qu' à l'itération i de l'algorithme 1, $islegal(src_i \gg dst_i, s_i) = faux$. Donc $(*)$ implique que la politique (D, s) est également violée.
3. S'il existe i tel que $d \in dst_i$ et src_i n'accède à aucun objet terminal d'aucun utilisateur du groupe G, alors la politique KR_p est violée si et seulement s'il existe $o \in O$ et $m \in M$ tels que $o.m \in src_i$ et la politique $KR_{p'}$ « G n'interfère pas avec o » est violée.
   Par hypothèse, la violation de $KR_{p'}$ entraîne l'existence d'un flux d'informations utilisant au moins un objet appartenant à $term(u)$ en tant qu'objet source et au moins l'objet o en tant qu'objet destination. Dans ce cas, l'opération $src_i \gg dst_i$ crée par composition un flux depuis au moins un objet appartenant à $term(u)$ vers d. Par conséquent, si $KR_{p'}$ est violée, $(*)$ implique que :
   $$islegal(src_i \gg dst_i, s_i) = faux$$
   et la politique (D, s) est donc violée.
4. Nous avons donc montré que si la politique KR_p est violée, alors $(*)$ implique que la politique (D, s) est violée. Ceci démontre la validité du théorème. □

Nous avons donc montré que toute politique *KR* peut être représentée dans le cadre de notre modèle par un ensemble d'objets terminaux et une politique (D, s) telle que définie par le théorème B.2.1. Nous savons que toute violation de la politique *KR* entraîne automatiquement une violation de cette politique (D, s) et par conséquent peut être détectée par l'algorithme 1.

Par conséquent, toutes les intrusions susceptibles d'être détectées par l'approche *KR* constituent des attaques par délégation détectables par l'approche que nous proposons.