Intégration de l'utilisateur au contrôle d'accès : du processus cloisonné à l'interface homme-machine de confiance

Mickaël Salaün

THÈSE DE DOCTORAT DE TÉLÉCOM SUDPARIS

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Mickaël Salaün

Pour obtenir le grade de

DOCTEUR de TÉLÉCOM SUDPARIS

Sujet de la thèse :

Intégration de l'utilisateur au contrôle d'accès : du processus cloisonné à l'interface homme-machine de confiance

Soutenue le 2 mars 2018 devant le jury composé de :

Président :	Gaël Thomas	Télécom SudParis
Rapporteurs :	Michaël Hauspie	Université Lille 1, Sciences et Technologies
	Valérie Viet Triem Tong	CentraleSupélec
Examinateur :	Roland Groz	Grenoble INP, Ensimag
Directeur de thèse :	Hervé Debar	Télécom SudParis
Co-encadrante :	Marion Daubignard	ANSSI
Invités :	Mathieu Blanc	CEA
	Benjamin Morin	ANSSI

NNT : 2018TELE0006

Mise à jour du 6 avril 2018

Résumé

Cette thèse souhaite fournir des outils pour qu'un utilisateur puisse contribuer activement à la sécurité de son usage d'un système informatique. Les activités de sensibilités différentes d'un utilisateur nécessitent tout d'abord d'être cloisonnées dans des domaines dédiés, par un contrôle d'accès s'ajustant aux besoins de l'utilisateur. Afin de conserver ce cloisonnement, celui-ci doit être en mesure d'identifier de manière fiable les domaines avec lesquels il interagit, à partir de l'interface de sa machine.

Dans une première partie, nous proposons un nouveau mécanisme de cloisonnement qui peut s'adapter de manière transparente aux changements d'activité de l'utilisateur, sans altérer le fonctionnement des contrôles d'accès existants, ni dégrader la sécurité du système. Nous en décrivons une première implémentation, nommée StemJail, basée sur les espaces de noms de Linux. Nous améliorons ce cloisonnement en proposant un nouveau module de sécurité Linux, baptisé Landlock, utilisable sans nécessiter de privilèges. Dans un second temps, nous identifions et modélisons les propriétés de sécurité d'une interface homme-machine (IHM) nécessaires à la compréhension fiable et sûre du système par l'utilisateur. En particulier, il s'agit d'établir un lien entre les entités avec lesquelles l'utilisateur pense communiquer, et celles avec lesquelles il communique vraiment. Cette modélisation permet d'évaluer l'impact de la compromission de certains composants d'IHM et d'aider à l'évaluation d'une architecture donnée.

Mots-clefs : sécurité, système d'exploitation, contrôle d'accès, activité utilisateur, cloisonnement, interface homme-machine (IHM), serveur d'affichage, chemin de confiance

Involving the end user in access control: from confined processes to trusted human-computer interface

Abstract

This thesis aims to provide end users with tools enhancing the security of the system they use. First, user activities of different sensitivities require to be confined in dedicated domains by an access control fitting the user's needs. Next, in order to maintain this confinement, users must be able to reliably identify the domains they interact with, from their machine's interface.

In the first part, we present a new confinement mechanism that seamlessly adapts to user activity changes, without altering the behavior of existing access controls nor degrading the security of the system. We also describe a first implementation named StemJail, based on Linux namespaces. We improve this confinement tool by creating a new Linux security module named Landlock which can be used without requiring privileges. In a second step, we identify and model the security properties a human-computer interface (HCI) requires for the reliable and secure understanding of the system by the user. Precisely, the goal is to establish a link between the entities with which the users think they communicate, and those with which they actually communicate. This model enables to evaluate the impact of HCI components jeopardization and helps assessing a given architecture.

Keywords: security, operating system, access control, user activity, confinement, sandboxing, human-computer interface (HCI), display server, trusted path

Table des matières

Table des figures

Remerciements

Tout d'abord, je remercie Hervé Debar, Benjamin Morin et Marion Daubignard pour leur encadrement. Ensuite, je souhaite remercier Michaël Hauspie et Valérie Viet Triem Tong qui m'ont fait l'honneur de rapporter cette thèse, ainsi que l'ensemble des membres du jury.

Je tiens notamment à remercier Loïc et Yves-Alexis pour m'avoir permis de travailler sur cette thèse au sein du Laboratoire architectures matérielles et logicielles (LAM) de l'Agence nationale de la sécurité des systèmes d'information (ANSSI).

Je souhaite également remercier Vincent pour son travail impressionnant qui a insufflé les principes fondateurs de mes travaux.

La rédaction d'article demande de nombreuses relectures et les commentaires qui en résultent forgent le document final. Je remercie toutes les personnes qui ont ainsi participé à ce manuscrit, ce qui inclut Alain, Anaël, Anne-Marie, Arnaud E., Arnaud F., François, Fred, Guillaume, Johan, Nico, Philippe, Ryad, Sylvie et Thomas.

Merci aux personnes qui ont bien voulu m'accorder de leur temps et me permettre de préparer ma soutenance de thèse, notamment Corentin, Frédéric, Nicolas B., Nicolas G., Rémi, Sébastien, Timothée et Tony.

Je remercie également celles et ceux qui me font le plaisir d'assister à ma soutenance et qui m'ont encouragé (et diverti) pendant ces quelques années.

Enfin, je tiens à dire un grand merci à Marion, pour son soutien, son enseignement et son aide inestimable durant cette longue période de doctorat.

Introduction

Le contrôle d'accès d'un ordinateur protège des données qui peuvent provenir de différentes sources comme le système de fichiers, le réseau, mais également l'utilisateur de la machine. La catégorisation de ces données par le système est nécessaire à la bonne application d'une politique de sécurité qui est chargée de restreindre leur accès. Que ce soit la rédaction d'un document ou la saisie d'un mot de passe, les entrées utilisateur peuvent correspondre à différents niveaux de sensibilité ainsi qu'à de multiples cercles de diffusion de l'information. Lorsqu'une politique de sécurité prend en compte ces différentes catégories, l'utilisateur est alors responsable de la différenciation des données qu'il fournit au système. Il prend donc implicitement part au contrôle d'accès. En tant que sujet capable d'accéder à différentes catégories de données, l'utilisateur est également une cible à protéger des programmes malveillants.

Une session utilisateur correspond à l'environnement logiciel que manipule l'utilisateur et dans lequel il évolue, communément après authentification sur le système. La menace prise en compte dans cette thèse est l'exécution d'une application malveillante dans une session utilisateur. Le comportement malveillant peut provenir du développeur de l'application, ou de la compromission de l'exécution d'une application via une donnée contrôlée par un attaquant. En d'autres termes, l'attaquant est capable d'exécuter du code sur la machine de l'utilisateur, ce qui lui permet d'accéder aux données mises à disposition par le système. Cet accès permet de récupérer des informations, de les modifier ou encore de les supprimer.

Afin que l'utilisateur puisse protéger ses données, il est nécessaire qu'il puisse indiquer ses intentions au système de contrôle d'accès, et ce pour chacune de ses activités. Pour ce faire, l'utilisateur doit connaître et comprendre les éléments les plus sensibles du système pour valider les accès légitimes à ses données. Ce contrôle d'accès défini par l'utilisateur est additionnel au contrôle d'accès géré par un administrateur qui a pour but de protéger le système et les utilisateurs entre eux.

Cette thèse a pour objectif de contribuer à la sécurité des systèmes informatiques pour la protection de l'utilisateur et de ses données. Une telle problématique comporte de multiples aspects, dont deux en particulier sont traités dans cette thèse. La première partie du travail est relative à l'adaptation du contrôle d'accès aux besoins de l'utilisateur, tandis que la seconde partie propose une modélisation de la sécurité d'une interface homme-machine (IHM) et son intégration dans un contrôle d'accès.

Protection des données utilisateur par le système

Le modèle d'accès discrétionnaire usuel s'applique à l'utilisateur sans tenir compte du niveau de sensibilité des données propre à chaque exécution d'application. Une application malveillante peut donc permettre à un attaquant d'accéder à toutes les données d'un même utilisateur, indépendamment de leur niveau de sensibilité.

L'application du principe de moindre privilège impose de restreindre les applications utilisées à ne pouvoir accéder qu'aux seules données strictement nécessaires à leur bon

fonctionnement. Dans ce but, cette thèse s'intéresse à raffiner la politique de sécurité en fonction de l'activité de l'utilisateur.

Un exemple d'application du concept est la pratique dite du *bring your own device* (BYOD) qui consiste à utiliser un même ordinateur, en pratique une même session utilisateur, pour un ensemble d'activités professionnelles et personnelles. Le cas d'un professionnel qui manipule des données appartenant à plusieurs de ses clients dans une même session fournit un exemple plus complexe. Le résultat de la compromission d'une application utilisée pour un de ses clients ne doit pas remettre en cause l'intégrité ou la confidentialité des données des autres clients. De telles attaques incluent l'exécution d'un rançongiciel (*ransomware*) qui chiffre les données, ou d'un logiciel d'exfiltration de données. Le contrôle d'accès proposé est capable de circonscrire de telles compromissions en cloisonnant des applications dans des environnements dédiés appelés domaines de sécurité. Cette approche permet de limiter l'impact d'une attaque à un sous-ensemble de données.

La configuration d'un contrôle d'accès peut être fastidieuse et nécessiter une expertise du système concerné. Un utilisateur final doit pourtant être capable d'exprimer son besoin au système et d'utiliser un contrôle d'accès sans que celui-ci ne le gêne dans son travail quotidien. Le modèle de contrôle d'accès proposé permet, une fois les accès autorisés initialement définis, de déterminer automatiquement l'activité courante et le périmètre qui y est légitimement accessible. Ce contrôle d'accès s'intègre au flux des tâches (*workflow*) de l'utilisateur pour supprimer la charge cognitive liée à des demandes intempestives de prise de décision de sécurité. Il en résulte un contrôle d'accès dynamique particulièrement bien adapté pour l'utilisateur qui permet de raffiner des restrictions liées à son activité. Le modèle est validé par une preuve de la conservation de ses propriétés de sécurité tout au long de l'évolution du contrôle d'accès pour une activité donnée.

StemJail [1, 2] constitue une mise en œuvre de ce modèle pour un système GNU/-Linux. Ce logiciel tire parti des espaces de noms du noyau Linux (*namespaces*) et plus particulièrement de l'espace de noms utilisateur pour créer des domaines de sécurité. Cette approche permet de mettre en place un cloisonnement évolutif accessible à un utilisateur qui ne dispose pas de privilèges d'administration. *StemJail* est développé en Rust [3] en suivant de bonnes pratiques de développement, ce qui permet d'avoir des garanties sur la stabilité et la sécurité de la solution. Enfin, la configuration des contrôles d'accès et de leur fonctionnement est abordable pour un utilisateur non expert.

Bien que pouvant convenir à nos besoins initiaux, il est apparu des limitations à l'utilisation des espaces de noms Linux, notamment en ce qui concerne l'expressivité des accès autorisés. En réponse à ce constat, nous avons développé un module de sécurité appelé *Landlock* [4, 5, 6, 7] qui repousse les limites des espaces de noms, en préservant la flexibilité du contrôle d'accès, sans pour autant nécessiter de privilèges d'administration. *Landlock* permet de valider les accès aux ressources du noyau Linux, particulièrement les fichiers, pour un ensemble défini d'applications. Il est également possible de raffiner le contrôle d'accès par la composition de plusieurs règles de sécurité. Cette approche donne aux développeurs d'applications les moyens de les isoler avec un contrôle d'accès au plus près du découpage logique interne (*sandboxing*), comme peuvent le faire certains navigateurs web. De la même manière, un logiciel comme *StemJail* peut utiliser cette fonctionnalité pour permettre à l'utilisateur de cloisonner ses activités plus finement.

Interaction de confiance entre l'utilisateur et le système

L'intégration de l'utilisateur au contrôle d'accès repose également sur les échanges d'informations entre l'utilisateur et le système. Cette communication s'effectue au travers de l'interface homme-machine, qui correspond à l'ensemble des périphériques matériels et des

composants logiciels destinés à transmettre les actions de l'utilisateur et à lui représenter des informations provenant du système. Cette interface assiste l'utilisateur pour la création, la visualisation et la modification de ses données. Afin que l'utilisateur puisse faire des choix respectant la politique de sécurité, il est nécessaire qu'il interprète correctement le contrôle d'accès à travers l'IHM. De même, pour valider la provenance de ces choix, une interaction intègre entre l'utilisateur et le système doit être garantie par l'IHM. Étant donnée son importance pour la bonne compréhension et l'utilisation du système, l'IHM est critique pour une intégration sécurisée de l'utilisateur au contrôle d'accès.

Cette thèse s'intéresse aux interfaces visuelles, qui comprennent principalement les interfaces graphiques et textuelles permettant d'interagir avec une session utilisateur. Pour étudier ces interfaces, il faut s'intéresser à des éléments logiciels tels que les serveurs d'affichage, les gestionnaires de fenêtres, les applications utilisateur, mais également le fonctionnement des composants matériels tels que les cartes graphiques ou encore les claviers. Les attaques par piégeage du matériel ou canaux auxiliaires sont cependant exclues de notre modèle d'attaque. De même, l'atteinte à la disponibilité est exclue de notre problématique.

Que ce soit la manipulation de l'affichage ou la capture des frappes clavier, les attaques qui tirent parti de l'IHM peuvent impacter l'intégrité et la confidentialité des données qui transitent par ce média. Ceci peut également permettre de manipuler l'utilisateur et de détourner la finalité de ses actions. Dans le cas de l'administrateur, de telles attaques peuvent donc mettre à mal tout le contrôle d'accès du système qu'il est chargé de maintenir.

Pour concevoir une IHM apte à protéger l'utilisateur et son interaction avec le système, nous proposons une modélisation des propriétés de sécurité attendues d'une IHM de confiance. Cette modélisation porte sur l'intégrité et la confidentialité des échanges, mais aussi sur ce que l'utilisateur comprend du système qu'il manipule, en particulier de ses fonctionnalités critiques comme le contrôle d'accès. Par exemple, il est nécessaire pour l'utilisateur de connaître les domaines de sécurité qu'il voit et ceux qui reçoivent ses frappes clavier. Ce modèle utilise la représentation des domaines de sécurité du système par des domaines abstraits pour l'utilisateur. Ceci permet d'exprimer des relations entre la perception du contrôle d'accès qu'a l'utilisateur et le contrôle d'accès réellement implémenté par le système.

Le travail de formalisation aboutit à l'établissement de conditions suffisantes pour une interaction de confiance entre un système sécurisé et l'utilisateur. Une IHM de confiance vérifiant ces conditions apporte donc des propriétés de sécurité qui permettent à un utilisateur averti d'interagir de manière sécurisée avec le système.

Pour vérifier les conditions suffisantes établies, il suffit d'utiliser une zone graphique de confiance apte à représenter de manière sécurisée les domaines avec lesquels l'utilisateur interagit. Cette zone de confiance est une mise en œuvre d'un chemin de confiance (*trusted path*) pour l'IHM, garantissant la véracité et l'intégrité des informations présentées. Ces informations comprennent les domaines qui utilisent les périphériques physiques de l'IHM, que ce soit pour afficher des données à l'utilisateur ou pour récupérer des données qu'il transmet via cette interface. Ces résultats théoriques permettent d'évaluer notre implémentation d'une barre de confiance dans le système sécurisé Clip OS [8].

Contributions et contenu de la thèse

Ce travail de thèse est centré sur la protection de l'utilisateur et des mécanismes jugés nécessaires pour assurer cette tâche. Les contributions sont réparties selon deux axes.

La première partie propose des mécanismes de cloisonnement de différentes activités utilisateur afin de circonscrire une compromission.

Le chapitre 1 présente l'état de l'art de la recherche, ainsi que des solutions qui mettent en place les fondements nécessaires à notre étude sur le contrôle d'accès.

Le chapitre 2 propose un nouveau modèle d'intégration transparente du contrôle d'accès en fonction de l'activité de l'utilisateur, particulièrement adapté pour un utilisateur non expert. Dans une session utilisateur, chaque application est exécutée sous l'identité de l'utilisateur. Une application peut donc accéder à toutes les données dont l'utilisateur est propriétaire. Dans ce cas, si l'information en elle-même est la cible de l'attaque, un adversaire n'a pas besoin d'élever ses privilèges et la politique de sécurité du système autorisera ce type d'accès dans notre modèle. Les actions de l'utilisateur sont utilisées pour associer dynamiquement une politique de sécurité à un groupe de processus, identifié comme faisant partie d'une activité utilisateur. Ce suivi permet d'effectuer des transitions d'un domaine de sécurité à l'autre, ce qui permet d'évoluer vers le domaine de sécurité final correspondant à une activité précise. Cette découverte d'activité permet de raffiner les restrictions d'un contrôle d'accès sans l'intervention explicite de l'utilisateur, ce qui a l'avantage de ne pas l'interrompre dans son travail et donc d'améliorer l'adoption de comportements sécurisés. Ces travaux ont été présentés à la conférence AsiaCCS [2].

Le chapitre 3 décrit *StemJail*, implémentation d'un contrôle d'accès fondé sur la détection des activités utilisateur. Cette nouvelle solution de cloisonnement *open source* valide expérimentalement l'utilisation des espaces de noms utilisateur Linux pour du cloisonnement non privilégié. Une gestion du contrôle d'accès qui ne nécessite pas de privilèges d'administration peut ainsi être obtenue, ce qui exclut toute escalade de privilèges liée à l'utilisation de *StemJail*. En plus de profiter d'une intégration transparente, la définition de la politique de sécurité est simplifiée pour permettre à un utilisateur non expert d'exprimer ses activités via l'organisation de ses fichiers par activité. Cette preuve de concept a été présentée aux conférences SSTIC [1] et AsiaCCS [2].

Après l'étude des limites de cette première solution, le chapitre 4 aborde *Landlock*, un contrôle d'accès *open source* pour Linux dédié au *sandboxing* non privilégié d'applications. Ce mécanisme permet de mettre en place une politique de sécurité dynamique pour un ensemble de processus. Une règle *Landlock* permet de valider chaque accès à un objet noyau, par exemple un fichier, ce qui apporte une granularité plus fine et une configuration plus dynamique que l'utilisation des espaces de noms Linux. Ce composant permet aux développeurs d'applications d'intégrer une politique de sécurité au plus proche de la logique applicative, ce qui autorise une intégration des mécanismes mis en œuvre par *StemJail* au cœur d'applications complexes comme des navigateurs web. Des travaux préliminaires sur *Landlock* ont été présentés à la conférence Kernel Recipes [4]. Nous avons également présenté aux conférences SSTIC [5], Linux Security Summit [6] et FOSDEM [7] l'avancée des développements, et un travail d'intégration dans la version de référence de Linux est en cours.

La seconde partie concerne la sécurité offerte par l'IHM, en tant que prolongement nécessaire du contrôle d'accès mis en œuvre par le système jusqu'à l'utilisateur.

Le chapitre 5 explique dans un premier temps les différents éléments qui composent une interface homme-machine et plus particulièrement une interface graphique. Une liste des menaces sur les composants d'IHM est établie, ce qui permet de définir des critères de sécurité sur les entrées et sorties utilisateur. Nous identifions ensuite différentes techniques de défense existantes. Enfin, un panorama liste et compare différentes approches de solutions de sécurité pertinentes qui prennent en compte la problématique de l'IHM.

Le chapitre 6 identifie et modélise les propriétés de sécurité nécessaires à la mise en œuvre d'une IHM de confiance. Comme le travail présenté en première partie, la modélisation repose sur le découpage en domaines système des éléments implémentés dans l'IHM. Ces domaines sont abstraits par l'utilisateur, qui construit sa propre vision du système. Dans

cette thèse, nous proposons la première formalisation complète des liens qui doivent exister entre les domaines réels et leur représentation par l'utilisateur pour vérifier différentes propriétés de sécurité que nous définissons également. Après avoir proposé une formalisation pour vérifier que ces propriétés de sécurité sont bien satisfaites, nous détaillons des stratégies d'implémentation d'IHM qui vérifient les critères formalisés. Enfin, nous illustrons notre approche avec une évaluation de l'IHM du système Clip OS et de notre implémentation d'une barre de confiance.

Première partie

Protection des données utilisateur par le système

Chapitre 1

État de l'art du contrôle d'accès

Sommaire

Ce chapitre passe en revue quelques modèles théoriques ainsi que des solutions de contrôle d'accès pertinentes pour notre sujet. La section 1.1 rappelle différents modèles de contrôle d'accès et leurs particularités principales. La section 1.2 propose un panorama de différentes mises en œuvre de contrôle d'accès par des systèmes d'exploitation ainsi que des outils qui en tirent parti. Enfin, la section 1.3 propose une comparaison des solutions citées à l'aune des propriétés nécessaires à la réalisation de notre objectif.

1.1 Modèles communs de contrôle d'accès

Le contrôle d'accès fait référence à la méthode utilisée pour autoriser ou refuser un accès d'un sujet à un objet, par exemple un processus à un fichier. L'application d'un tel contrôle est assurée par un moniteur de référence, rôle communément joué par le noyau d'un système d'exploitation. Une définition plus formelle correspondant à notre approche sera donnée dans la section 2.2. Il existe plusieurs modèles de contrôle d'accès parmi lesquels nous retrouvons ceux présentés ci-après.

1.1.1 *Discretionary Access Control* (DAC)

Le contrôle d'accès discrétionnaire [9] permet à un sujet d'appliquer sa propre politique de sécurité sur ses objets. C'est le contrôle d'accès classiquement utilisé dans les systèmes de type UNIX comme GNU/Linux. L'administrateur du système peut généralement outrepasser les accès des autres utilisateurs. Il n'est cependant pas responsable de la bonne configuration du contrôle mis en place par les autres utilisateurs.

1.1.2 *Mandatory Access Control* (MAC)

Le contrôle d'accès obligatoire [9] permet d'imposer une politique de sécurité centralisée, appliquée à tous les sujets et les objets d'un système. Contrairement au DAC, le MAC ne permet pas à chaque sujet de définir ses propres règles. C'est usuellement l'administrateur du système qui configure ou modifie la politique de sécurité du système, ce qui doit par ailleurs être autorisé par cette même politique.

Un contrôle d'accès de type MAC peut être configuré pour appliquer différents modèles de politiques de sécurité, comme le modèle Bell et LaPadula [10], qui reflète des restrictions militaires pour prévenir des fuites de données en fonction de leur niveau de sensibilité. À l'inverse le modèle Biba [11] permet de protéger l'intégrité des objets en fonction de leur niveau de sensibilité. La politique de sécurité de la Grande Muraille (*Chinese wall*) [12] constitue un autre exemple permettant de gérer des conflits d'intérêts.

1.1.3 *Role-Based Access Control* (RBAC)

Le contrôle d'accès basé sur des rôles [13] est un modèle de sécurité qui arbitre l'accès à des ressources via des structures abstraites appelées rôles. Un rôle permet d'associer un sujet à un ensemble de droits. Une décision d'accès est prise en fonction du rôle qu'un sujet joue. Un sujet peut être autorisé à changer de rôle parmi une liste de rôles permis. Le principe est que le sujet se place dans le rôle correspondant à la tâche qu'il souhaite effectuer, et bénéficie ainsi des droits appropriés. Cette approche simplifie particulièrement la gestion des accès pour un nombre important de sujets et d'objets.

1.1.4 *Domain Type Enforcement* (DTE)

Le modèle DTE [14, 15] permet de définir des domaines qui caractérisent des contextes d'exécution restreints par des règles de contrôle d'accès. Un sujet, privilégié ou non, peut

ainsi être cloisonné à un domaine donné. Les contraintes qui s'appliquent dans son domaine d'exécution permettent de protéger le reste du système des accès définis comme illégitimes. Le DTE offre notamment les outils nécessaires pour mettre en œuvre du MAC.

1.1.5 Capacités de sécurité (*capabilities*)

Une capacité de sécurité, à ne pas confondre avec une capacité POSIX, peut être définie comme un jeton non falsifiable et transmissible. Comme le décrivent Shapiro *et al.* [16], ce jeton est associé à une liste d'actions autorisées sur les données qu'il référence. La sécurité basée sur les capacités consiste d'une part à concevoir des logiciels aptes à se transmettre des capacités de manière à respecter le principe de moindre privilège, et d'autre part à avoir l'infrastructure nécessaire à ces partages au niveau du système d'exploitation. Ce système de capacités simplifie la gestion des accès, mais nécessite que chaque sujet soit compatible avec ce paradigme.

1.1.6 *Sandbox*

Une *sandbox* est définie par Schreuders *et al.* [17] comme un contrôle d'accès axé sur le fonctionnement légitime d'une application, indépendamment des droits de son utilisateur. Ce contrôle d'accès peut être implémenté par le développeur de l'application dans celle-ci. La confiance dans ce contrôle d'accès est donc relative à la confiance dans l'origine et l'intégrité de l'application.

1.2 Solutions de contrôle d'accès fournis par le système d'exploitation

1.2.1 Conteneurs

Les conteneurs sont des environnements dits de virtualisation légère, habituellement utilisés pour héberger des services. Le noyau de l'hôte est en charge de chaque environnement cloisonné, pour ce qui concerne le réseau, le système de fichiers ou encore la visibilité des processus. Que ce soient les *jails* de FreeBSD, les *zones* de Solaris, les cages Linux-VServer, les conteneurs OpenVZ ou les espaces de noms et *cgroups* de Linux, ces solutions de cloisonnement sont mises en place et gérées par l'administrateur d'un système. Les conteneurs ne sont pas orientés utilisateur et n'ont pas de configuration dynamique qui pourrait s'adapter au changement d'activité de celui-ci.

1.2.2 Contrôles d'accès obligatoires de Linux

Comparés par Schreuders *et al.* [17], les mécanismes de contrôle d'accès obligatoire dédiés à Linux tels que SELinux, AppArmor, Tomoyo ou Smack forment un groupe de moniteurs faisant partie du noyau. Ces solutions peuvent mettre en place du contrôle d'accès de type DTE, ce qui permet notamment d'appliquer du MAC ou du RBAC pour créer et gérer finement des politiques de restriction de processus. Ces politiques complètent le contrôle d'accès discrétionnaire traditionnel (DAC) et nécessitent des privilèges d'administration.

Plus particulièrement, le composant SELinux [18] est une implémentation de *Flux Advanced Security Kernel* (FLASK). Il fournit les mécanismes nécessaires pour assurer la confidentialité et l'intégrité des différents objets du système. Certains services en espace utilisateur peuvent également tirer parti de ces fonctionnalités pour appliquer eux-mêmes un contrôle d'accès à leur niveau. En raison du caractère obligatoire de sa politique de

sécurité, un contrôle d'accès obligatoire ne permet pas à un simple utilisateur de définir lui-même des contraintes que l'administrateur n'aurait pas prévues.

1.2.3 Android et iOS

Les systèmes d'exploitation récents tels qu'Android et iOS fournissent les moyens à l'utilisateur du système de contrôler certains accès de ses applications. L'utilisateur n'a pas besoin de privilèges d'administration pour imposer des restrictions d'accès à ses données. Les accès contrôlés sont de haut niveau et compréhensibles par l'utilisateur, par exemple l'accès aux contacts ou aux photos. Par contre, l'utilisateur n'est pas capable d'exprimer une politique de sécurité précise et flexible pour contrôler l'utilisation des fichiers entre applications. Les applications utilisateur sont conçues spécifiquement et ne peuvent être installées et exécutées que sur le système pour lequel elles ont été développées.

1.2.4 XNU Sandbox

XNU Sandbox [19], précédemment appelé Seatbelt, est un composant de sécurité utilisé par macOS et iOS, basé sur l'infrastructure TrustedBSD [20]. Il permet d'exprimer des règles en syntaxe *S-expression* avec des mots-clefs définis permettant à un logiciel de s'appliquer à lui-même des limitations sur les accès aux fichiers ou encore au réseau. Les fichiers sont identifiés par leur chemin via des expressions rationnelles (automate fini non déterministe). L'utilisation d'une telle *sandbox* ne nécessite pas de privilèges d'administration.

1.2.5 *seccomp-bpf*

Le mécanisme de sécurité *seccomp-bpf* [21] n'est pas à proprement parler un mécanisme de *sandboxing*, mais permet de réduire la surface d'attaque exposée par le noyau en limitant les appels système autorisés. En effet, comme l'indique sa documentation [22], il agit comme un pare-feu entre l'espace utilisateur et l'espace noyau, qui est configurable sans nécessiter de privilèges d'administration. Le filtrage peut être effectué sur le numéro de l'appel système ainsi que sur ses arguments sous forme de valeurs de 64 bits. Par exemple, il est possible de filtrer l'appel système `open` ainsi que les drapeaux utilisés pour l'ouverture de fichier comme `O_RDONLY`, ce qui permet d'y limiter l'accès en lecture seule. En revanche, étant donné que le chemin du fichier à ouvrir est exprimé sous forme d'un pointeur vers une chaîne de caractères, seule cette adresse peut être filtrée, ce qui n'est donc pas suffisant pour identifier le chemin de fichier référencé.

1.2.6 Capsicum

Le projet Capsicum [23] a pour but d'implémenter le principe des capacités dans un système UNIX. Pour ce faire, les descripteurs de fichier sont étendus pour leur ajouter des contraintes d'accès, les transformant en capacités. De la même manière que pour les fichiers, Capsicum apporte également un type descripteur équivalent pour les processus. Ces capacités sont un moyen, pour les applications compatibles avec cette fonctionnalité, d'exposer leurs ressources à d'autres processus tout en restreignant finement et efficacement leurs usages.

Afin de contrôler l'usage de certains services du système, par exemple la résolution de noms réseau, le *framework* et démon Casper [24] permet d'utiliser des fonctionnalités de haut niveau en se reposant sur les capacités offertes par Capsicum. En effet, certaines fonctions qui ont du sens pour l'espace utilisateur sont inconnues du noyau. Son but est de fournir des fonctionnalités bas niveau utilisées par l'espace utilisateur pour traiter les données correspondant habituellement aux couches 5 à 7 du modèle OSI.

1.2.7 Pledge

Pledge [25] est un système de *sandboxing* minimaliste développé et utilisé par OpenBSD. Il permet de décrire un ensemble d'autorisations via des mots-clefs. Ces autorisations regroupent l'utilisation d'ensembles d'appels système ayant une sémantique proche, comme par exemple `rpath` qui permet d'accéder aux fichiers en lecture seule. Pledge est spécifique au système OpenBSD et son implémentation fait des hypothèses fortes sur l'organisation du système de fichiers. Par exemple, l'utilisation de listes blanches de chemins de fichiers comme `/etc/localtime` est inscrite en dur dans le noyau.

1.2.8 Mbox

Mbox [26] est un outil de *sandboxing* qui fonctionne en utilisant la fonctionnalité de débogage `ptrace` tout en minimisant les appels système grâce à un filtre *seccomp-bpf*. Cette approche permet d'obtenir un système de cloisonnement qui ne nécessite pas de privilèges d'administration tout en offrant de meilleures performances que les *sandboxes* antérieurs. De la même manière que d'autres projets comme Janus [27] ou Systrace [28], il utilise un moniteur qui valide les requêtes des processus cloisonnés. Mbox interagit avec l'utilisateur pour faire évoluer les droits d'accès, ce que nous retrouvons d'ailleurs dans Alcatraz [29], une autre solution similaire.

Le premier défaut de cette approche est que le contrôle d'accès a lieu entièrement en espace utilisateur ce qui impacte les performances, notamment à cause des changements de contexte fréquents de chaque processus cloisonné. Le deuxième inconvénient important est que le contrôle d'accès repose sur une émulation du comportement du noyau. Comme l'expose Garfinkel [30], cette approche est périlleuse et peut facilement engendrer des vulnérabilités du fait de *race conditions*, d'erreurs d'implémentation du comportement ou de synchronisation d'états avec le système d'exploitation. Pour ces raisons, nous avons exclu les solutions d'émulation du noyau pour le contrôle d'accès.

1.3 Comparaison des mises en œuvre de contrôles d'accès

Tout utilisateur peut être amené à exécuter du code malveillant, qu'il provienne d'une application non de confiance ou d'une application initialement de confiance, mais compromise. La mise en place d'un contrôle d'accès est un prérequis à la protection des données d'un utilisateur vis-à-vis d'applications malveillantes. Nous souhaitons donner la possibilité à chaque utilisateur de configurer un contrôle d'accès obligatoire pour un ensemble de sujets qui le représentent. Les restrictions peuvent concerner l'accès aux systèmes de fichiers, au réseau ou encore aux communications inter-processus. Le contrôle d'accès mis en place par l'administrateur du système doit cependant être prioritaire et donc respecté à tout moment. Les quatre propriétés fonctionnelles suivantes ont été identifiées pour répondre à ce besoin.

1.3.1 Accessibilité à tous les utilisateurs

Pour répondre aux besoins de chaque utilisateur d'un système, nous recherchons un contrôle d'accès qui puisse être accessible et configurable par n'importe quel utilisateur, sans privilèges d'administration. Chaque utilisateur doit être en mesure de personnaliser son contrôle d'accès afin qu'il puisse l'utiliser et soit pertinent.

Le DAC est accessible à tous les utilisateurs, mais ne leur permet pas de retirer définitivement un ensemble d'autorisations à un ensemble de processus.

Les conteneurs permettent de créer un environnement restreint, mais leur création nécessite aujourd'hui des privilèges d'administration. Par exemple, la création et l'accès aux

périphériques de type *bloc* posent des problèmes d'intégrité et de confidentialité des données qui y sont stockées. Le système Linux n'est pas prévu pour déléguer de telles ressources à des utilisateurs potentiellement non de confiance.

Le MAC, et plus particulièrement SELinux, est dédié à l'administrateur. Une seule politique est définie pour tout le système et il n'est pas possible de déléguer partiellement sa personnalisation sans impacter la sécurité globale du système.

1.3.2 Finesse du contrôle d'accès

Un contrôle d'accès efficace doit permettre d'exprimer finement les sujets, les objets et les actions. Ceci est particulièrement pertinent pour l'accès aux fichiers de l'utilisateur par ses applications. Les instances de celles-ci doivent donc être des sujets à part entière.

Les systèmes Android et iOS permettent à l'utilisateur de valider certaines permissions accordées à leurs applications. Ces permissions sont cependant restreintes et fixées par le système : l'utilisateur ne peut ni en ajouter ni les personnaliser.

seccomp-bpf n'effectue pas à proprement parler du contrôle d'accès, mais plutôt de la réduction de surface d'attaque du noyau. Bien que cette fonctionnalité puisse être détournée pour restreindre des accès, *seccomp-bpf* ne permet pas d'identifier des objets noyau.

1.3.3 Mise à jour dynamique de la configuration

Un contrôle d'accès dédié à l'utilisateur doit pouvoir être mis à jour à la volée pour s'adapter aux changements d'activité de l'utilisateur de manière transparente.

Le DAC, SELinux, *XNU sandbox*, *seccomp-bpf* et Pledge sont prévus pour appliquer une politique de sécurité de manière statique. Bien qu'il soit possible de faire évoluer les droits associés à un sujet, par exemple par une transition de rôle, il n'est cependant pas possible de faire évoluer a posteriori la définition des sujets et des accès qui leur sont autorisés.

1.3.4 Compatibilité avec les applications existantes

De manière pragmatique, la compatibilité d'une solution de contrôle d'accès avec les applications actuellement utilisées par un système, sans requérir leur modification, est nécessaire pour une utilisation effective sur un système existant. Bien que les applications mériteraient d'avoir moyen de se *sandboxer*, la modification de code existant ne doit pas être un prérequis à l'utilisation d'un contrôle d'accès.

Le contrôle d'accès d'Android et d'iOS doit en partie être pris en compte par les applications utilisateur, notamment par le respect d'API et d'IPC pour accéder à certaines ressources.

Capsicum et Pledge sont conçus pour être intégrés dans les applications. Il est cependant difficile, voire impossible, d'utiliser ces fonctionnalités sur des applications non adaptées.

1.3.5 Récapitulatif

Le tableau 1.1 résume la comparaison entre différents systèmes majeurs qui peuvent être utilisés pour contraindre des applications utilisateur. Nous constatons qu'aucune des solutions existantes ne satisfait les besoins intrinsèques au contrôle d'accès que nous avons établis : un contrôle d'accès accessible à tous les utilisateurs, d'une granularité fine, avec la possibilité de mettre à jour dynamiquement sa configuration, et qui soit compatible avec les applications existantes. Les chapitres suivants détaillent notre approche du contrôle d'accès adapté à l'utilisateur et proposent une solution théorique et pratique.

	Référence	**Accessibilité**	**Finesse**	**Mise à jour**	**Compatibilité**
conteneurs	1.2.1		∼		✓
SELinux	1.2.2		✓		✓
Android/iOS	1.2.3	✓		✓	∼
XNU/Sandbox	1.2.4	✓	✓		✓
seccomp-bpf	1.2.5	✓			✓
Capsicum	1.2.6	✓	✓	✓	
Pledge	1.2.7	✓			

TABLE 1.1 – Comparaison de contrôles d'accès

Chapitre 2

Contrôle d'accès adapté aux activités utilisateur

Sommaire

Ce chapitre explique notre démarche pour la modélisation d'un contrôle d'accès évolutif en fonction des actions de l'utilisateur. Dans la section 2.1, nous commençons par décrire un cas d'usage que nous utiliserons pour illustrer notre modèle. Ensuite, la section 2.2 définit formellement le vocabulaire que nous utilisons autour de la notion d'activité. La section 2.3 justifie la mise en place d'un cloisonnement par activité et en identifie les écueils potentiels. Nous diagnostiquons le besoin de découvrir l'activité utilisateur, qui est détaillée en section 2.4. La section 2.5 formalise notre modèle en se basant sur des automates, ce qui permet d'expliciter et de prouver les garanties de sécurité qu'il fournit. Enfin, la section 2.6 traite des limites de notre approche et des solutions envisagées.

2.1 Scénario d'exemple

De nombreux cas d'usage peuvent illustrer le besoin de partitionnement de données. Nous prenons l'exemple d'un consultant nommé *Bob* qui gère les comptes de plusieurs de ses clients, notamment *GoodGuy* et *BadGuy*. *Bob* communique avec ses clients par une application de messagerie électronique grâce à laquelle il reçoit des documents pour la gestion de leurs comptes. Chaque ouverture de document correspond à une nouvelle instanciation de l'application *OfficeSheet*.

Bob est un utilisateur non privilégié de son ordinateur et ne peut donc ni gérer les accès du système ni interférer avec d'autres utilisateurs. L'administration de la machine est laissée à l'administrateur dont le rôle est de protéger le système et les utilisateurs entre eux. Par contre, cet administrateur n'est ni en responsabilité ni en capacité de s'assurer que les processus des utilisateurs ne sont pas malveillants. Bien qu'il puisse être difficile pour un utilisateur d'identifier tous les comportements légitimes, nous définissons un comportement comme malveillant lorsqu'il va à l'encontre de la volonté de l'utilisateur. Le postulat est que l'utilisateur est le mieux à même de définir les accès légitimes de ses applications *à ses données*.

BadGuy est un client légitime de *Bob* mais souhaite profiter de cette situation pour récupérer des informations sur son concurrent *GoodGuy*. Pour cela il conçoit des documents malveillants dont l'ouverture avec *OfficeSheet* provoque l'exécution de commandes à travers celui-ci. Ces documents peuvent simplement contenir des macros malveillantes ou exploiter une vulnérabilité de l'application *OfficeSheet*.

Bob souhaite se protéger des conséquences de la compromission d'une application comme *OfficeSheet*. Dans ce scénario, deux cibles sont identifiées : les données des clients et celles de *Bob*. L'accès aux données en lecture impacte leur confidentialité, alors que leur accès en écriture met à mal leur intégrité. Afin de limiter les dommages causés par une application qui deviendrait malveillante, *Bob* souhaite définir des règles simples d'utilisation de ses données par instance d'application. Les données de ses clients ne doivent être accessibles par *OfficeSheet* que lors de leur utilisation légitime. Il peut cependant y avoir un certain nombre de données qui doivent être légitimement accédées ou partagées entre plusieurs clients. En plus de ses clients, *Bob* gère également son compte bancaire *MyBank* dans la même session utilisateur et utilise la même application *OfficeSheet* pour ouvrir ses documents bancaires.

Trois activités émergent naturellement de cet exemple simple mais réaliste. Les fichiers nécessaires pour travailler pour le client *GoodGuy* (respectivement le client *BadGuy* ou *MyBank*) appartiennent à une activité que nous appelons *GoodGuy* (respectivement *BadGuy* ou *MyBank*). Une solution de cloisonnement devrait fournir à l'utilisateur les moyens de travailler sur ces trois activités simultanément. Un changement rapide d'activité doit être possible, par exemple pour répondre à une demande urgente.

Bien que requérant de l'utilisateur *Bob* un certain niveau de connaissance de l'outil informatique, notre approche ne devrait pas nécessiter un niveau d'expertise en sécurité des

systèmes d'information. La séparation entre les activités telle qu'introduite ici correspond à l'idée que peut se faire un utilisateur standard et qu'il est capable de définir par lui-même. Le seul prérequis supposé est que l'utilisateur d'une solution de cloisonnement doit être capable de spécifier des cercles de diffusion d'information, c'est-à-dire des réseaux d'interlocuteurs pour des sujets donnés.

Nous considérons que l'utilisateur ne devrait pas être chargé de marquer chaque fichier avec les activités qui y sont liées, démarche fastidieuse et propice aux erreurs. Nous considérons comme trop incertaine une solution à base d'inférence entièrement automatisée des activités en fonction du comportement global de l'utilisateur. Nous proposons un compromis entre ces deux approches permettant de mettre à profit la compréhension de l'utilisateur pour obtenir une solution de cloisonnement adaptée. Concrètement, l'organisation des fichiers dans des dossiers correspondant à des activités semble suffisamment simple à initier et à garder cohérente au fil du temps. Dans notre exemple, nous supposons que notre utilisateur a créé un dossier `~/Clients/GoodGuy/` (respectivement `~/Clients/BadGuy/` et `~/Accounts/`) où il stocke les fichiers liés à *GoodGuy* (respectivement *BadGuy* et *MyBank*). Notre hypothèse est que les données contenues dans chaque dossier sont liées à l'activité qui donne son nom au dossier.

2.2 Définitions formelles

La notion d'activité est subjective : nous pouvons la définir comme un ensemble de tâches cohérentes du point de vue de l'utilisateur. Nous introduisons des définitions permettant de formaliser la politique de contrôle d'accès que nous souhaitons voir appliquer pour isoler les activités entre elles. Chacune des activités que l'utilisateur peut effectuer dans une même session doit lui permettre d'accéder à un ensemble de ressources associées.

2.2.1 Rôle

Comme décrit par Ferraiolo et Kuhn [13], un *rôle* fait référence à une fonction donnant autorité et légitimité pour effectuer les tâches relatives à ce rôle. Le besoin d'isolation vient du fait que les utilisateurs jouent souvent plusieurs rôles. Le concept d'activité utilisateur introduit précédemment peut naturellement se formaliser comme un rôle défini par l'utilisateur. Dans notre exemple, l'utilisateur *Bob* a un rôle dédié quand il travaille pour un client spécifique, c'est-à-dire le rôle *GoodGuy* ou le rôle *BadGuy*.

Plus généralement, de la même manière que plusieurs accès peuvent être légitimes pour plusieurs activités, il est possible que l'utilisateur exerce différentes disjonctions de rôles qu'il a définis. Nous les appelons rôles intermédiaires. Par exemple, le rôle intermédiaire « *GoodGuy ou BadGuy* » représente des tâches indifféremment exécutées par le rôle *GoodGuy* ou le rôle *BadGuy*.

2.2.2 Objet

En prenant le même vocabulaire que RBAC, décrit dans la section 1.1.3, un *objet* de notre politique est une donnée utilisateur prise en compte par notre politique de contrôle d'accès. Dans le cadre de cette thèse, un objet de type fichier est représenté par ce qui est plus couramment appelé un chemin de fichier. Il identifie un dossier (et son contenu) ou un fichier du système de fichiers. Les données référencées par un tel objet peuvent donc apparaître, évoluer ou disparaître au cours du temps.

2.2.3 Action et accès

Les *actions* désignent des opérations sur des objets. Nous utilisons particulièrement deux actions, `read` et `write`, qui désignent les opérations de lecture et d'écriture sur les ressources associées, mais notre modèle peut être étendu avec d'autres actions.

Nous définissons un *accès* comme le fait d'effectuer une action donnée sur un objet.

2.2.4 Règle

Une *règle* est un triplet qui consiste en un rôle, un objet et une action. De manière informelle, quand un utilisateur spécifie une règle (r, o, a), il exprime qu'il devrait être légitimement capable d'effectuer l'action a sur l'objet o quand il joue le rôle r. À la vue des éléments introduits précédemment, nous nous attendons à ce que notre utilisateur ait besoin d'écrire des données dans les fichiers stockés dans `~/Clients/GoodGuy/` quand il travaille pour son client *GoodGuy*. Ceci est exprimé par la règle (*GoodGuy*, `~/Clients/GoodGuy/`, `write`).

2.2.5 Domaine

Les *domaines utilisateur* sont définis par les utilisateurs et sont constitués d'un ensemble de règles communes à un même rôle. Comme nous l'avons vu, une activité utilisateur est formalisée par un rôle du point de vue de l'utilisateur. Pour chacun de ces rôles, l'utilisateur liste les règles qui définissent les actions appropriées sur les objets lorsqu'il exerce l'activité correspondant à ce rôle. Chacune de ces listes produit une configuration dédiée qui définit un domaine. Il peut donc y avoir plusieurs domaines utilisés en parallèle. Par ailleurs, il est crucial que l'utilisateur comprenne l'isolation qu'il veut voir appliquer, afin de définir correctement ses domaines.

Les *domaines intermédiaires* sont formés par les intersections des domaines utilisateur. Les domaines intermédiaires sont les homologues des rôles intermédiaires : intuitivement, si une règle apparaît dans deux domaines utilisateur, un processus respecte cette règle quand il se conforme à l'un comme à l'autre de ces domaines utilisateur.

Dans ce chapitre, un *domaine de sécurité*, abrégé en *domaine*, est donc utilisé pour désigner une politique de sécurité qui s'appliquera sur un contexte d'exécution via un contrôle d'accès. Dans les chapitres suivants, le mot *domaine* sera utilisé pour désigner l'environnement résultant de l'application d'une politique. Dans tous les cas, ce concept de contexte d'exécution restreint par une politique apparaît comme la clef de voûte des différentes parties de notre travail.

2.2.6 Sujet et moniteur

Un *sujet* est un processus soumis aux règles d'un domaine. Un utilisateur peut donc être représenté simultanément par plusieurs sujets.

Un *moniteur* est une entité de référence, par exemple le noyau du système d'exploitation ou un processus, chargée de contrôler les actions effectuées par les sujets afin qu'ils se conforment aux règles d'un domaine.

2.3 Politique de sécurité pour l'utilisateur

Dans le cadre de notre problématique, l'utilisateur est déjà restreint par le contrôle d'accès du système, que ce soit celui local à la machine ou celui imposé sur les accès aux fichiers partagés sur le réseau. Nous souhaitons lui donner les moyens de se protéger de certaines instances de ses applications qui pourraient avoir un comportement malveillant, que ce soit initialement ou au cours de leur exécution. Une politique de sécurité n'a pas

vocation à décider de l'innocuité d'une application, mais plutôt à appliquer un ensemble de règles qui peuvent refuser des actions identifiées comme inappropriées.

Nous nous orientons donc vers du contrôle d'accès obligatoire, mais pour lequel d'une part l'administration de la politique est l'apanage de l'utilisateur plutôt que de l'administrateur de la machine, et d'autre part les contraintes sont imposées à des processus plutôt qu'aux utilisateurs de la machine. Bien entendu, ce contrôle d'accès ne doit en aucun cas remettre en cause la politique de sécurité mise en place par l'administrateur. L'utilisateur est seulement capable d'ajouter des contraintes supplémentaires.

Cette dernière exigence n'est cependant pas suffisante pour s'assurer que la superposition d'un contrôle d'accès supplémentaire soit inoffensive pour le système existant. En effet, imposer des contraintes sur un sujet peut modifier son comportement d'une manière non prévue lors de sa conception. Par exemple, sous Linux, un binaire SUID peut créer un processus avec des droits différents de l'utilisateur qui l'exécute. Ces droits apportent souvent des privilèges pour l'utilisateur courant, comme c'est le cas lors de l'exécution d'un binaire SUID *root* ou avec une capacité particulière. Des changements de droits d'accès provoqués par un utilisateur ne doivent pas impacter le comportement de sujets qui ont des droits supérieurs à ceux de l'utilisateur. Un utilisateur ne doit donc pas être en mesure d'appliquer ses propres contraintes à des sujets autres que les siens, afin de prévenir toute escalade de privilèges.

Dans notre cas, c'est l'utilisateur final qui décrit les politiques de sécurité qui s'appliquent à ses instances d'application, car il est chargé des données qu'il crée, pour son propre usage, celui de ses clients, ou encore de ses collaborateurs. C'est donc lui le plus à même de caractériser les accès légitimes. Chaque activité peut impliquer différents intervenants, qui forment autant de cercles de diffusion de l'information. L'accès aux données peut également être de nature différente, par exemple en lecture seule ou en lecture et écriture. Ces propriétés identifient des rôles uniques qu'il est nécessaire de bien distinguer pour la protection des données. Par conséquent, chaque activité va être associée à un rôle et un domaine formé par les règles de contrôle d'accès légitimes dans le cadre de son exercice. Ainsi, notre solution pourra mettre en place un domaine par activité, cloisonnant de ce fait les activités entre elles selon les cercles de diffusion pertinents pour l'utilisateur.

Une fois les rôles identifiés par l'utilisateur, il faut associer un rôle à un sujet, que ce soit de manière explicite ou implicite. L'association explicite de rôle fait intervenir l'utilisateur pour chaque choix de rôle que doit faire le système. C'est par exemple le cas pour l'*User Account Control* [31] (UAC) de Windows qui demande explicitement l'accord de l'utilisateur pour donner les droits demandés par un sujet, autrement dit, de changer le rôle du processus exécuté. Comme le mettent en avant Furnell et Thomson [32], ce type de demande, souvent via des fenêtres modales, peut être sujet à des incompréhensions ou des effets d'habitude et de fatigue, ce qui peut mener à des choix malencontreux.

À l'inverse, une association implicite de rôles vise à ne pas faire intervenir l'utilisateur au moment où le contrôle a lieu mais en amont, ce qui lui permet de ne pas être interrompu lorsqu'il est concentré sur une tâche particulière. Cette approche a pour but d'intégrer de manière transparente la sécurité dans le flux des tâches de l'utilisateur, autrement appelé *workflow* de l'utilisateur. Une réponse partielle à l'intégration dans ce *workflow* est d'adapter les applications et le système d'exploitation comme le fait le modèle *User-Driven Access Control* qui sera détaillé dans la section 5.3.6. Les accès alors appliqués sont cependant statiques et ne répondent pas à la problématique du cloisonnement des activités utilisateur.

Aussi, nous proposons dans notre solution de rendre cette association à la fois implicite et dynamique, ce qui nous amène à introduire la découverte automatisée du rôle joué par les sujets.

2.4 Découverte automatisée de rôles

Une solution de sécurité dédiée à l'utilisateur devrait lui être la moins intrusive possible pour avoir un périmètre d'utilisation plus large et donc améliorer davantage l'impact d'une telle solution. Il est donc préférable de limiter les choix utilisateur au fur et à mesure de son utilisation du système en prenant en compte ses réflexions et choix passés. Pour assigner un rôle à un sujet, sans interaction avec l'utilisateur, tout en restant sécurisé, notre moniteur de référence doit être capable de limiter autant que possible les accès des sujets tout en trouvant le rôle correspondant le mieux à l'activité utilisateur. Le but ici est de ne pas nécessiter d'actions supplémentaires de l'utilisateur pour tout rôle qu'il a précédemment défini. Nous devons donc être capables de découvrir les rôles d'un sujet en fonction de ses actions.

Pour chaque sujet, le moniteur utilise un état pour stocker un domaine, appelé domaine courant, qui correspond au contrôle d'accès actuellement appliqué au sujet. Le principe consiste à démarrer l'exécution d'un sujet avec un domaine courant dénué d'objets accessibles, et à procéder à l'ajout d'accès possibles à la volée. Le moniteur doit alors avoir une stratégie pour décider si une demande d'accès d'un sujet doit être accordée. Cette stratégie doit garantir que si un sujet demande des accès autorisés par (au moins) un domaine utilisateur donné tout au long de son exécution, le sujet obtient effectivement les autorisations sollicitées. La stratégie doit également garantir que seules de telles séries d'accès sont autorisées. Ceci revient à autoriser les actions acceptées par au moins un domaine utilisateur qui autorise les actions déjà effectuées.

Quand un sujet commence son exécution, il est associé à un domaine courant vide ; trois cas peuvent se produire. Sa première demande d'accès sur un objet peut être absente de tous les domaines utilisateur, apparaître dans un seul domaine utilisateur, ou enfin dans plusieurs d'entre eux. Dans le premier cas, il est aisé d'affirmer que la requête doit être interdite. Le deuxième cas est également assez simple : il donne la capacité au moniteur de déterminer à quel domaine utilisateur correspond l'activité. L'état du moniteur doit être mis à jour pour correspondre à ce domaine utilisateur, et toutes les requêtes subséquentes doivent se conformer aux spécifications de ce domaine particulier. Le troisième cas est probablement le plus courant, mais également le plus complexe. La requête doit être accordée, mais l'état du moniteur ne peut pas associer un domaine utilisateur particulier au domaine courant.

C'est ici qu'interviennent les domaines intermédiaires. Ces domaines correspondent aux actions sur des objets présents dans de multiples domaines utilisateur. Dans le cas présent, le moniteur doit évoluer pour avoir comme domaine courant le domaine intermédiaire correspondant à l'intersection de tous les domaines utilisateur qui contiennent la demande d'accès requise par le sujet. Ce domaine intermédiaire correspond aux différents rôles possibles de l'utilisateur et permet de transiter vers leurs domaines utilisateur parents. De cette manière, le moniteur identifie que l'activité courante fait partie de celles correspondant à ces domaines utilisateur.

Au cours du reste de l'exécution du sujet, pour chaque nouvelle autorisation d'accès à un objet, le domaine courant ne doit être autorisé à évoluer que vers un de ces domaines utilisateur. Nous tenons ainsi compte de l'historique des accès demandés par le sujet. Dans le cas contraire, il serait possible qu'aucun domaine utilisateur n'autorise l'ensemble des accès aux objets effectués par le sujet, ce qui est censé être la garantie de sécurité.

La stratégie du moniteur est ici implémentée sous forme d'un automate basé sur les domaines intermédiaires. Une transition entre un domaine intermédiaire source et destination existe si le domaine destination contient le domaine source. Ceci formalise le fait que tous les accès jugés légitimes dans le domaine source doivent également l'être dans le domaine destination.

La figure 2.1 illustre une séquence possible de transitions de domaines. Comme décrit

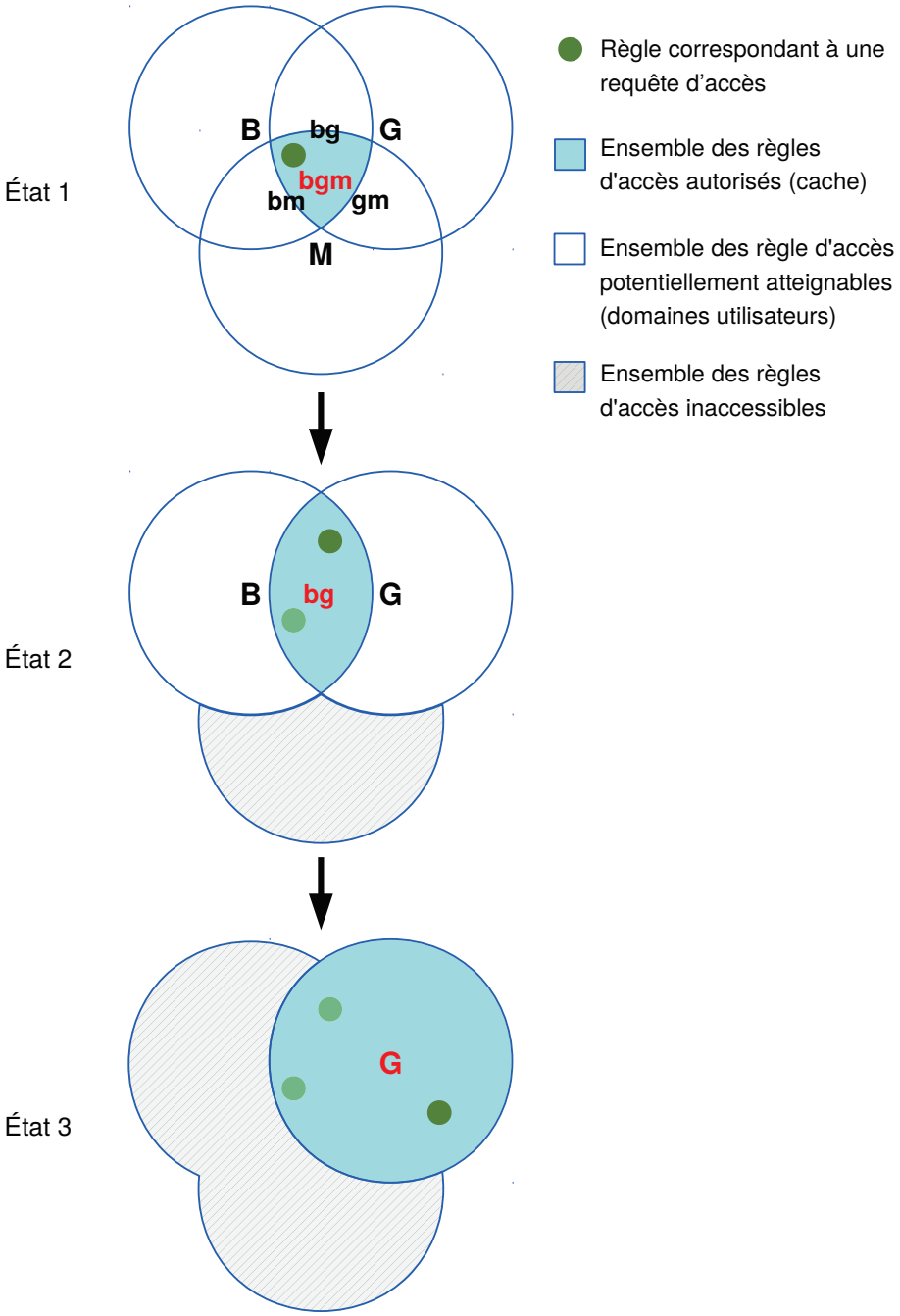


FIGURE 2.1 – Exemple de spécialisation de domaine par transition

dans la section 2.1, il y a trois domaines utilisateur : *GoodGuy* (`G`), *BadGuy* (`B`) et *MyBank* (`M`). Certains objets et accès sont autorisés par des règles figurant dans plusieurs domaines. Ceci est illustré par les chevauchements qui forment un domaine intermédiaire : `bg`, `bm`, `gm` et `bgm`. La figure 2.2 montre toutes les transitions possibles selon la stratégie mise en œuvre. Le déroulement du scénario est le suivant.

1. Un sujet demande initialement un accès à un objet commun aux trois domaines. Le point vert foncé symbolise la règle correspondante dans le domaine intermédiaire formé par l'intersection des trois domaines : `bgm` (*BadGuy ou GoodGuy ou MyBank*). À cette étape, tous les domaines intermédiaires sont potentiellement atteignables, ce qui veut dire que tous leurs fichiers sont potentiellement accessibles par les sujets.

2. Lorsqu'un accès est accordé pour un objet par une règle dans les domaines *GoodGuy* et *BadGuy*,mais pas dans le domaine *MyBank*, le moniteur effectue une transition de `bgm` vers `bg` (*GoodGuy ou BadGuy*). Cette transition est autorisée car le domaine d'origine (`bgm`) est un sous-ensemble du domaine destination (`bg`). Le domaine courant contient alors un nouvel ensemble de règles qui autorisent le sujet à effectuer l'accès qu'il a précédemment demandé. Nous pouvons noter que par souci d'optimisation, le sujet peut utiliser un cache pour limiter les requêtes d'accès et ne contacter le moniteur que lorsqu'une transition est requise. L'état du moniteur traduit le fait que le rôle courant n'est pas lié au domaine *MyBank*, et qu'aucun accès à un objet uniquement autorisé dans ce domaine ne sera dorénavant accordé.
3. Quand un sujet demande l'accès à un objet, tel que décrit par une règle dans *BadGuy*, mais non présent dans `bg`, le moniteur fait transiter le domaine courant vers *BadGuy*, qui est un domaine utilisateur. Les accès précédemment demandés et accordés sont toujours autorisés, mais il n'y a plus de transition possible.

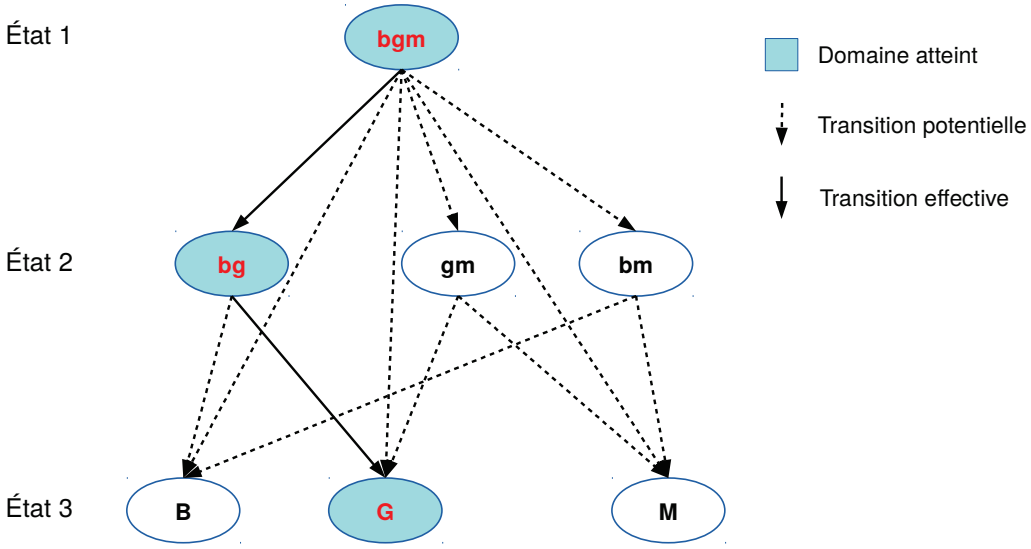


FIGURE 2.2 – Treillis de transitions de domaine

Nous pouvons souligner que la découverte de rôle ne requiert pas que tous les domaines utilisateur aient une intersection commune. En revanche, une requête d'accès initiale doit nécessairement correspondre à au moins une règle d'un domaine utilisateur pour être acceptée. Les requêtes suivantes seront gérées séquentiellement. Quand une requête d'accès ne correspond à aucune règle d'un domaine accessible, l'accès est refusé. Le moniteur reste dans le domaine courant et le sujet ne sera pas en mesure d'accéder à l'objet.

À ce stade de la présentation de la stratégie de découverte de rôle, nous pouvons également souligner que suivant l'ordre dans lequel les accès sont effectués par un sujet, un accès donné peut être autorisé ou non. En d'autres termes, il n'y a pas de résultat absolu de l'autorisation d'accès à un objet, celle-ci va dépendre de la séquence d'accès demandés précédemment. Tant qu'il est encore vraisemblable que nous soyons dans le cadre d'un rôle (intermédiaire ou non) défini par l'utilisateur, l'accès est autorisé.

Si un sujet est compromis alors que son moniteur l'autorise à transiter vers de multiples domaines, alors toutes les ressources disponibles dans les domaines accessibles peuvent également être compromises. Par exemple, dans le cas où le domaine `bgm` est compromis, alors les ressources des autres domaines sont compromises. Pour profiter au mieux des propriétés de sécurité de notre modèle, la définition des domaines doit suivre les lignes directrices suivantes. Pour chaque domaine utilisateur, les données critiques doivent être

stockées dans des dossiers dédiés au domaine. Ce faisant, un accès à ces données fait évoluer le domaine courant vers le domaine utilisateur approprié et assure que ces données ne sont utilisées que dans le cadre d'une activité légitime. De plus, la politique de sécurité doit interdire l'accès en écriture aux données partagées entre de multiples domaines, ce qui correspond à un canal de communication inter-domaine.

Dans le pire des cas, le nombre de domaines intermédiaires vaut $2^n - 1$ pour n domaines utilisateur. En pratique, cette complexité spatiale exponentielle n'est pas un problème pour le moniteur étant donné le faible espace mémoire requis par domaine et le nombre relativement limité de domaines utilisateur fortement similaires dans notre cas d'usage.

2.5 Formalisation et garanties du modèle

2.5.1 Ordre partiel des domaines

Nous introduisons dans notre formalisation un ordre partiel pour comparer des domaines. Nous souhaitons formaliser l'idée que la politique sous-jacente d'un domaine peut être plus restrictive que celle d'un autre domaine. L'ensemble des domaines possibles est noté D et un de ses éléments est noté d. L'ensemble des domaines utilisateur est noté U et est composé de f éléments, chacun noté u. Étant donné que U est un ensemble fini, nous pouvons écrire $U = \{u_1, \ldots, u_f\}$.

Nous commençons par ordonner les objets. Étant donné deux arborescences de fichier o et o' dans un ensemble O, nous écrivons $o \sqsubseteq o'$ si et seulement si o' est identique à, ou parent de o. Intuitivement, cela veut dire que o fait référence à un fichier ou un dossier qui est inclus dans le dossier pointé par o'. La relation $\sqsubseteq$ est une relation d'ordre car elle respecte les propriétés suivantes :

- réflexivité : $\forall o \in O, o \sqsubseteq o$ car un objet est identique à lui-même ;
- antisymétrie : $\forall o, o' \in O, (o \sqsubseteq o' \wedge o' \sqsubseteq o) \Rightarrow o = o'$ car si o' est parent de o et o est parent de o', alors o et o' sont identiques ;
- transitivité : $\forall o, o', o'' \in O, (o \sqsubseteq o' \wedge o' \sqsubseteq o'') \Rightarrow o \sqsubseteq o''$ car si o' est parent de o et o'' est parent de o', alors o'' est parent de o.

Les accès sont des paires constituées d'une action $a \in A$ et d'un objet $o \in O$. L'ordre partiel sur les objets peut être généralisé aux accès avec la notation $\preccurlyeq$ comme suit.

$$(a, o) \preccurlyeq (a', o') \Leftrightarrow a = a' \wedge o \sqsubseteq o'$$

Les propriétés de l'ordre $\preccurlyeq$ sont héritées de celles de l'ordre $\sqsubseteq$. L'ordre $\preccurlyeq$ reflète le fait que l'accès (a, o) est plus restrictif que l'accès (a', o'), étant donné que les éléments dans o sont compris dans o'. Nous pouvons souligner que les actions de lecture et d'écriture ne sont pas comparables.

Les règles sont des triplets contenant un rôle $r \in R$, une action $a \in A$, et un objet $o \in O$, alors que les accès ne représentent qu'une action et un objet. Un accès (a, o) est dit *conforme à un domaine* si et seulement s'il existe une règle du domaine qui représente un accès plus général. Cette conformité est exprimée avec la notation $\preccurlyeq$ comme suit.

$$(a, o) \preccurlyeq d' \Leftrightarrow \exists (r', a', o') \in d', (a, o) \preccurlyeq (a', o')$$

La décision d'un moniteur est alors formalisée par la fonction suivante, qui associe un booléen à un domaine et un accès.

$$\textsc{Access} : D \times A \times O \longrightarrow \mathbb{B}$$

$$(d, a, o) \longmapsto \begin{cases} \text{vrai si et seulement si } (a, o) \preccurlyeq d \\ \text{faux autrement} \end{cases}$$

Une liste d'accès est dite conforme à un domaine d si tous les accès de cette liste sont conformes à d.

Comparer des règles signifie comparer leur accès, sans tenir compte des rôles auxquels ils se rapportent : une règle en raffine une autre si les accès référencés par la première sont inférieurs à ceux référencés par la seconde. En d'autres termes, de deux règles, la plus grande est celle qui autorise le plus de droits d'accès lorsqu'un sujet s'y contraint. Nous pouvons finalement définir un ordre partiel pour comparer des domaines en tant qu'ensembles de règles. Ainsi, nous notons qu'un domaine d est inférieur à un domaine d', noté $d \trianglelefteq d'$, si toutes les règles de d raffinent des règles de d' :

$$d \trianglelefteq d' \Leftrightarrow \forall (r, a, o) \in d, \exists (r', a', o') \in d' \mid (a, o) \preccurlyeq (a', o')$$

Ceci s'écrit naturellement en termes de conformité des accès de d à d' :

$$d \trianglelefteq d' \Leftrightarrow \forall (r, a, o) \in d, (a, o) \preccurlyeq d'$$

2.5.2 Automate pour l'évolution des domaines

Nous choisissons de représenter le moniteur par un automate dont les états sont les domaines intermédiaires. De manière informelle, ces domaines intermédiaires consistent en des accès correspondant à plusieurs domaines utilisateur. Nous devons donc définir formellement l'intersection de deux domaines. Une telle intersection forme un nouveau domaine, correspondant à un rôle que nous appelons « r ou r' ». Une règle apparaît dans le nouveau domaine si l'accès qu'elle décrit est listé dans une règle de l'un des domaines d'intersection et s'il est inférieur aux accès listés dans les règles des autres domaines. Elle représente donc l'ensemble maximal des accès qui se conforment aux deux domaines. De manière formelle, l'intersection des domaines d et d', notée $d \sqcap d'$, correspond aux règles (r ou r', a_0, o_0) de telle sorte que :

— soit $(r, a_0, o_0) \in d$ et $(a_0, o_0) \preccurlyeq d'$,
— ou alors $(r', a_0, o_0) \in d'$ et $(a_0, o_0) \preccurlyeq d$.

Cette définition se généralise aisément au cas d'une intersection finie de domaines.

Toute règle d'une intersection de domaines est raffinée par au moins une règle de chaque domaine formant l'intersection. En conséquence, quand un moniteur autorise des accès qui correspondent à une intersection de domaines, nous pouvons légitimement déduire que les accès effectués par ce sujet respectent tous les domaines formant cette intersection.

L'ensemble $H \subseteq D$ des états de notre automate correspond à l'ensemble de toutes les intersections possibles des domaines utilisateur :

$$H = \left\{ \prod_{i \in J, u_i \in U} u_i, J \in \wp([1, f]) \right\}$$

L'état initial d_0 de l'automate est le domaine vide. Nous notons que d_0 est un élément de H, en tant qu'intersection d'aucun domaine utilisateur.

Nous pouvons maintenant passer à la fonction de transition entre les domaines telle qu'implémentée par le moniteur. En pratique, quand un sujet demande un accès à un objet, le moniteur doit décider s'il autorise la demande du sujet. Comme expliqué dans la section 2.4, cette décision dépend de la conformité de l'accès vis-à-vis du domaine correspondant actuellement à l'état du moniteur. Si c'est le cas, le sujet est autorisé à procéder. Dans le cas contraire, le moniteur vérifie s'il existe un domaine intermédiaire moins restrictif auquel l'accès pourrait correspondre. L'ensemble de tels domaines est décrit

avec la fonction suivante.

$$\begin{aligned}\textsc{Next} : H \times A \times O &\longrightarrow \wp(H) \\ (d, a, o) &\longmapsto \left\{ d' \in H \mid d \trianglelefteq d', \textsc{Access}(d', a, o) \right\}\end{aligned}$$

Si l'accès n'est dans aucun domaine intermédiaire, l'ensemble $\textsc{Next}(d, a, o)$ est vide mais le moniteur reste dans son état courant, ce que nous choisissons de formaliser par une transition laissant l'état invariant. Le moniteur refuse alors la demande d'accès du sujet. Si l'accès peut légitimement être effectué, le moniteur transite alors vers le domaine intermédiaire le moins restrictif parmi $\textsc{Next}(d, a, o)$, ce qui correspond à son élément minimal. Nous pouvons noter que si (a, o) correspond au domaine courant d, alors cet élément minimal est d. Ceci donne la définition suivante pour la fonction de transition.

$$\begin{aligned}\textsc{Transit} : H \times A \times O &\longrightarrow H \\ (d, a, o) &\longmapsto \textsc{Max}(\{d, \textsc{Min}(\textsc{Next}(d, a, o))\})\end{aligned}$$

Notation 1 (Système de transition d'états étiqueté)**.** Soit un système de transition d'états étiqueté (*Labelled Transition System* : LTS) $\mathcal{A} = \langle \Sigma, L, \rightarrow \rangle$, nous notons $\eta \xrightarrow{l} \eta'$ la transition de l'état $\eta \in \Sigma$ vers $\eta' \in \Sigma$ via une transition étiquetée avec $l \in L$.

L'automate d'évolution des domaines est défini sous forme d'un LTS. Une transition d'un état de l'automate à un autre est étiquetée par une action et un objet. L'ensemble de ces étiquettes est noté $L = A \times O$. L'automate est donc noté $\mathcal{A} = \langle H, L, \textsc{Transit} \rangle$. Nous utilisons la notation $d \xrightarrow{(a,o)} d'$ pour une transition de cet automate. De manière classique, un état d_i est dit *atteignable* si et seulement s'il existe une suite $d_0 \xrightarrow{(a_1,o_1)} d_1 \xrightarrow{(a_2,o_2)} \ldots \xrightarrow{(a_i,o_i)} d_i$ de transitions de l'état initial à d_i.

2.5.3 Garanties de sécurité prouvées

Lemmes utiles

Quand le moniteur transite d'un domaine à l'autre, tous les accès correspondant au domaine source restent valides pour le domaine cible. Cette propriété est formalisée en utilisant l'ordre partiel des domaines : les traces forment une chaîne ascendante.

Lemme 1 (Transition croissante)**.** *S'il existe une trace* $d_0 \xrightarrow{(a_1,o_1)} d_1 \xrightarrow{(a_2,o_2)} \ldots \xrightarrow{(a_i,o_i)} d_i$ *alors* $d_0 \trianglelefteq d_1 \trianglelefteq \cdots \trianglelefteq d_i$.

Démonstration. Étant donné $d'' \in \textsc{Next}(d, a, o)$, alors par définition nous avons $d \trianglelefteq d''$. De ce fait $d \trianglelefteq \textsc{Min}(\textsc{Next}(d, a, o))$. En conséquence $d \trianglelefteq \textsc{Max}(\{d, \textsc{Min}(\textsc{Next}(d, a, o))\}) = \textsc{Transit}(d, a, o)$. Enfin, comme $d \xrightarrow{(a,o)} d'$ dénote $d' = \textsc{Transit}(d, a, o)$, nous avons bien $d \trianglelefteq d'$. □

Chaque accès appartenant à un domaine intermédiaire, formé par l'intersection de plusieurs domaines, est conforme à tous les domaines formant cette intersection. L'intersection est alors inférieure à chacun des domaines la composant.

Lemme 2 (Intersection de domaines)**.** *Un domaine est toujours supérieur ou égal à son intersection avec un autre domaine :* $\forall d, d' \in D, (d \sqcap d') \trianglelefteq d$

Démonstration. Étant donné $(a, o) \in d \sqcap d'$, par définition de l'intersection de domaines, nous avons soit $(a, o) \preccurlyeq d$ et $(a, o) \in d'$, soit $(a, o) \preccurlyeq d'$ et $(a, o) \in d$. Dans les deux cas $(a, o) \preccurlyeq d$. □

Garantie formelle de sécurité

Nous pouvons maintenant établir le théorème garantissant le fait que le moniteur applique bien la politique de sécurité souhaitée. De manière intuitive, étant donné que le moniteur est mis en œuvre de telle manière qu'il ne transite qu'entre intersections de domaines utilisateur, il en résulte que les accès qu'il peut autoriser à ses sujets appartiennent forcément à des domaines utilisateur. Cependant, ce n'est pas suffisant : nous souhaitons que les sujets effectuent des accès se conformant au moins à un domaine utilisateur *tout au long de leur exécution*. En effet, toujours se conformer à un domaine utilisateur, mais pas nécessairement le même, ne fournit pas suffisamment de garanties. Cela revient à fournir un contrôle d'accès avec un domaine constant formé par l'union de tous les domaines utilisateur.

Bien qu'il ne puisse pas forcément déterminer ce domaine utilisateur au début de l'exécution, le moniteur doit cependant garantir que l'exécution d'un sujet respecte les restrictions du contrôle d'accès correspondant à au moins un domaine utilisateur.

Théorème 1 (Conformité à un domaine utilisateur)**.** *Étant donnée une trace* $d_0 \xrightarrow{(a_1,o_1)} d_1 \xrightarrow{(a_2,o_2)} \dots \xrightarrow{(a_n,o_n)} d_n$*, il existe un domaine utilisateur* $\Delta \in U$ *majorant tous les domaines de la trace :*

$$\exists \Delta \in U, \forall i \in [0,n], d_i \trianglelefteq \Delta$$

Démonstration. Nous considérons une trace de longueur n, $d_0 \xrightarrow{(a_1,o_1)} d_1 \xrightarrow{(a_2,o_2)} \dots \xrightarrow{(a_n,o_n)} d_n$. Le lemme 1 donne $d_0 \trianglelefteq d_1 \trianglelefteq \cdots \trianglelefteq d_n$. Deux cas peuvent se présenter :

- nous pouvons dans un premier temps avoir $d_n = \emptyset$. Il en découle que $d_0 = \cdots = d_n = \emptyset$. Dans ce cas, nous pouvons choisir n'importe quel domaine utilisateur $\Delta \in U$, puisque $\forall d \in D, \emptyset \trianglelefteq d$;
- dans le cas contraire, nous avons $d_n \neq \emptyset$. Étant donné que $d_n \in H$, nous savons que $d_n = \bigsqcap_{i \in J} u_i$ pour un ensemble $J \in \wp([0,f])$. De plus, étant donné que $d_n \neq \emptyset$, alors $J \neq \emptyset$. Nous définissons $\Delta = u_{i_0}$ pour $i_0 \in J$. Nous savons que $u_{i_0} \in U$ donc $\Delta \in U$. Dans le cas où $J = \{i_0\}$, nous avons $d_n = u_{i_0} = \Delta$. Autrement, étant donné que $d_n = u_{i_0} \sqcap (\bigsqcap_{i \in J, i \neq i_0} u_i)$, le lemme 2 nous donne $d_n \trianglelefteq u_{i_0} = \Delta$. La conclusion pour toute la chaîne s'ensuit : $d_0 \trianglelefteq d_1 \trianglelefteq \cdots \trianglelefteq d_n \trianglelefteq \Delta$.

□

2.6 Limitations du modèle de contrôle d'accès

L'approche décrite dans ce chapitre présente quelques inconvénients. En premier lieu, la compromission d'un sujet tôt dans son exécution peut permettre d'exploiter les domaines ainsi accessibles. Un moniteur dans un domaine intermédiaire est capable de transiter vers tous les domaines qui incluent le domaine intermédiaire. Un sujet malveillant a de ce fait potentiellement accès à toutes les données à disposition dans un domaine accessible par une transition. Si le domaine intermédiaire, dans lequel se trouve le sujet malveillant, autorise un accès en écriture à une ressource, cette ressource pourrait être utilisée comme point de collecte des données situées dans les autres domaines vers lesquels une transition est possible. Par exemple, si *OfficeSheet* est compromis dès le domaine `bg` (*GoodGuy ou BadGuy*) et que la politique l'autorise à ouvrir un dossier `~/Backup/` en écriture dans ce domaine, alors il pourrait dans un premier temps accéder aux données de *GoodGuy* et les copier dans `~/Backup/`, puis lors d'une deuxième exécution dans le domaine `bg`, accéder au domaine *BadGuy* tout en gardant légitimement un accès à `~/Backup/`. Il faut cependant noter qu'un tel scénario n'est possible que si la compromission de l'application persiste entre plusieurs

exécutions, ou si le vecteur de compromission est accessible aux différents domaines et qu'il infecte l'instance d'application avant spécialisation dans un de ces domaines. Comme précisé dans la section 2.4, la création d'une politique doit prendre en compte la circulation possible des données entre domaines.

Différentes activités telles que définies par un utilisateur peuvent se chevaucher. Si un domaine est strictement inclus dans un autre, le moniteur n'appliquera jamais la politique la plus restrictive. Par exemple, nous supposons qu'un domaine `B` nécessite un accès à `/a/b`, alors qu'un domaine `A` nécessite un accès à `/a`. Étant donné que l'accès à `/a/b` est possible dans les deux domaines, une requête pour un tel accès ne fera pas transiter le moniteur vers le domaine `B` une fois dans le domaine `A`. Une solution à ce problème serait de fournir plus d'informations au moniteur, via de nouveaux types de requêtes, pour compléter les chemins de fichiers. Ceci pourrait consister en une validation active provenant de l'utilisateur, dans ce cas très particulier seulement, comme l'implémentent l'UAC de Windows ou les systèmes de permission d'iOS et d'Android.

2.7 Conclusion

Nous avons décrit dans ce chapitre un nouveau modèle de contrôle d'accès qui permet d'appliquer automatiquement une séparation des informations en fonction des besoins utilisateur.

Notre objectif était d'être capable d'intégrer de manière transparente un contrôle d'accès pertinent dans le *workflow* utilisateur. Cette intégration a pour but d'améliorer et de simplifier l'utilisation d'une politique de sécurité par rapport aux contrôles d'accès traditionnels. Cet objectif est atteint par une découverte automatique de rôle qui permet de déduire progressivement l'activité d'un utilisateur et de transiter vers des domaines dédiés en conséquence. L'utilisateur n'a pas besoin de choisir initialement un rôle particulier, et la hiérarchie de classement de fichiers lui est déjà connue sans apprentissages supplémentaires.

Nous avons formalisé cette approche jusqu'à apporter une preuve de garanties de sécurité vérifiées tout au long de la vie d'un sujet.

Chapitre 3

StemJail : contrôle d'accès dynamique pour Linux

Sommaire

Dans ce chapitre, nous présentons l'implémentation du modèle présenté dans le chaptire 2 qui propose une séparation des activités utilisateur s'ajoutant au contrôle d'accès global du système. Nous supposons que le système d'exploitation utilisé par l'utilisateur est à l'état de l'art de la sécurité et est correctement configuré, et nous souhaitons fournir une solution pragmatique pour permettre aux utilisateurs d'accomplir leurs différentes activités de manière sécurisée.

Notre solution *open source* nommée *StemJail* [33] est une implémentation du contrôle d'accès défini dans le chapitre 2, pour un système GNU/Linux sans modifications intrusives. *StemJail* tire son nom des cellules souches (*stem cell*) capables de se multiplier, mais surtout de se spécialiser. Notre preuve de concept permet d'inférer automatiquement les activités de l'utilisateur en se basant sur ses actions. Pour implémenter une solution non invasive ne requérant pas de privilèges, nous avons fait le choix d'une architecture hybride : un gestionnaire de contrôle d'accès en espace utilisateur sert à définir et mettre à jour dynamiquement la politique de sécurité, conçue par l'utilisateur, mais appliquée par le noyau.

La section 3.1 énumère les objectifs fonctionnels et de sécurité que notre solution se doit de remplir. Viennent ensuite les explications liées à l'implémentation de *StemJail* dans la section 3.2, ce qui comprend l'architecture de ce système de cloisonnement, une présentation des mécanismes sous-jacents de Linux qui sont utilisés, les contraintes de développement ainsi que l'intégration et l'utilisation de la solution. La section 3.3 permet de valider notre solution avec le déroulement détaillé d'un scénario d'utilisation, une revue des problèmes potentiels de sécurité ainsi que des tests de performance. Par la suite, dans la section 3.3.4, nous effectuons un récapitulatif des solutions les plus pertinentes abordées dans le chapitre 1, soulignant les raisons pour lesquelles aucune d'entre elles ne répond complètement à la problématique que nous traitons. Enfin, la section 3.3.5 fait un point sur les limites de *StemJail* et les évolutions possibles.

3.1 Besoins fonctionnels et de sécurité

Définir des politiques de sécurité nécessite d'identifier clairement les parties de confiance du système, laissant le reste sous le contrôle potentiel d'un attaquant. Dans notre cas d'usage, la *Trusted Computing Base* (TCB) inclut le noyau du système d'exploitation, les services du système ainsi que le matériel. Les attaques par canaux auxiliaires ou canaux cachés sont en dehors du périmètre de cette thèse.

Le but de *StemJail* est de fournir aux utilisateurs un outil de contrôle des flux de données possibles entre les différentes activités en les cloisonnant entre elles. *StemJail* apporte donc un mécanisme permettant aux utilisateurs d'assurer la confidentialité et l'intégrité de leurs données liées à des activités spécifiques, par rapport à d'autres activités. En d'autres termes, confiner une activité nous permet de circonscrire les conséquences d'une compromission d'activité par un attaquant.

En plus du but principal de *StemJail*, les propriétés abordées dans la section 1.3 se concrétisent en six contraintes fonctionnelles et de sécurité qui doivent être remplies pour proposer une solution pertinente et de qualité.

Besoin 1 (Intégration au *workflow* utilisateur)**.** Pour être efficace, une infrastructure de contrôle d'accès dédiée à l'utilisateur final ne doit pas l'interrompre. En étant intégré dans le flux des tâches de l'utilisateur, *StemJail* applique un contrôle d'accès de manière transparente pour l'utilisateur.

Besoin 2 (Accessibilité à tous les utilisateurs)**.** Pour répondre aux besoins de chaque utilisateur d'un système, *StemJail* doit être accessible et configurable par n'importe quel

utilisateur, sans privilèges d'administration. Chaque utilisateur personnalise la configuration de *StemJail* pour qu'il soit utilisable dans sa session utilisateur.

Besoin 3 (Compréhensibilité par l'utilisateur)**.** Pour être capables d'utiliser correctement *StemJail*, ses utilisateurs doivent comprendre le cloisonnement mis en place. Ils devraient donc n'avoir qu'à influencer le contrôle d'accès de telle sorte qu'il ait un sens pour eux.

Besoin 4 (Innocuité)**.** L'ajout d'une nouvelle fonctionnalité de sécurité comme *StemJail* ne doit pas dégrader le niveau de sécurité du système. En particulier, nous ne modifions pas la TCB, et surtout pas le noyau, pour ne pas introduire de nouvelles vulnérabilités de sécurité. Les fonctionnalités de contrôle d'accès mises en œuvre dans *StemJail* n'ont pas pour but de remplacer une politique de sécurité configurée par les administrateurs du système. Elles doivent compléter celles déjà présentes sur un système correctement administré.

Besoin 5 (Compatibilité avec les applications existantes)**.** La compatibilité avec les applications actuellement utilisées, sans nécessiter leur modification, améliore considérablement l'aspect pratique de notre solution. *StemJail* doit pouvoir être utilisé avec une distribution GNU/Linux standard.

Besoin 6 (Impact minimal sur les performances)**.** Enfin, il est courant qu'une solution de sécurité ait un impact sur les performances du système. Cependant, pour qu'une solution soit réellement utilisable en pratique, cet impact doit être minimal.

3.2 Implémentation de *StemJail*

3.2.1 Aperçu de l'architecture

L'architecture de *StemJail* est conçue pour s'adapter de manière dynamique à de multiples activités utilisateur, chacune d'elles étant confinée dans un environnement dédié appelé *cage*. Comme le montre la figure 3.1, il y a trois composants principaux : un portail, un moniteur par cage et un ou plusieurs sujets par cage.

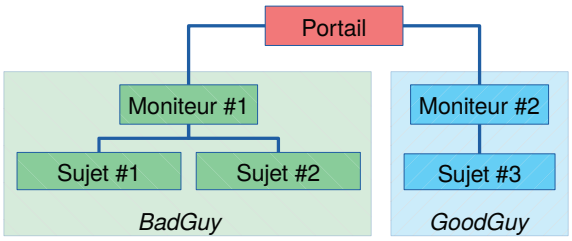


FIGURE 3.1 – Aperçu d'une instanciation de *StemJail*

Le portail est responsable de la création d'une cage et du lancement d'un moniteur avec lequel il garde un canal de communication pour lui envoyer des commandes et recevoir des informations sur son état.

Chaque moniteur est le processus initial d'une cage et est donc dédié à cette cage. Le moniteur d'une cage est le seul processus privilégié dans cet environnement : c'est le seul capable d'étendre les accès de la cage en fonction de la politique de sécurité.

Les sujets sont des instances d'applications utilisateur pour une activité donnée. Dans une cage, chaque sujet est un client capable d'effectuer des demandes d'ajout d'un nouvel accès au moniteur. C'est le seul moyen pour un sujet d'acquérir de nouveaux accès étant donné qu'il est non privilégié et le restera. En effet, nous nous reposons sur le noyau, qui fait partie de la TCB, pour éviter toute escalade de privilège. Les détails concernant le fonctionnement de ces mécanismes seront abordés dans la section 3.2.4.

Dans notre exemple, la cage *BadGuy* contient deux sujets capables d'envoyer des requêtes vers leur moniteur, qui implémente actuellement la politique de sécurité du domaine *BadGuy*. Ces trois processus peuvent communiquer entre eux, mais sont isolés de la cage *GoodGuy* et de ses sujets. Cette seconde cage peut avoir des données privées protégées de la vue de la première cage.

Grâce à cette architecture, *StemJail* est capable de créer autant de cages que nécessaire tout en les gérant de manière cohérente.

3.2.2 Les espaces de noms Linux

Comme expliqué dans la section 3.1, *StemJail* doit être capable d'appliquer des restrictions sur la visibilité des données, mais nous nous astreignons cependant à ne pas modifier le noyau ou nécessiter de privilèges. Pour ce faire, *StemJail* utilise les espaces de noms Linux (*namespaces*) [34]. Chacun d'eux expose un ensemble de fonctionnalités du noyau à l'espace utilisateur. Ils peuvent être utilisés pour isoler des processus les uns des autres. Ceux qui nous intéressent sont les espaces de noms utilisateur, *mount*, UTS, IPC, PID et réseau. L'espace de noms *cgroups* n'est actuellement pas utilisé par *StemJail* et ne sera donc pas abordé.

Mount

L'espace de noms *mount* permet de créer des points de montage isolés du système de fichiers, seulement visibles des processus de l'espace utilisateur qui se situent dans cet espace de noms. Ces points de montage exposent une hiérarchie de fichiers correspondant à une partition dédiée provenant d'un périphérique de stockage ou d'une sous-partie de la hiérarchie de fichiers d'un point de montage. En effet, Linux permet de monter un système de fichiers à partir de son répertoire racine, mais également à partir d'un autre sous-répertoire, via un *bind-mount*. Avec cet espace de noms, nous sommes donc capables de limiter la vue du système de fichiers principal pour certains processus.

UTS

L'espace de noms UTS tire son nom d'*UNIX Time-sharing System*. Il permet de personnaliser le nom de la machine à un sous-ensemble de processus.

IPC

L'espace de noms de communications inter-processus (*Inter-Process Communication*) permet d'isoler les communications via System V : *message queues*, sémaphores et mémoire partagée.

PID

L'espace de noms PID peut être utilisé pour créer une nouvelle numérotation de processus locale à un sous-ensemble de processus, mais permet surtout de cacher les processus en dehors de cet espace de noms à ceux qui y appartiennent. Les processus extérieurs à un tel espace de noms ne sont pas atteignables via les appels système de manipulation de processus tels que `signal` qui permet par exemple de tuer un processus.

Réseau

L'espace de noms réseau permet de créer un réseau isolé, ce qui comprend les périphériques réseau, les tables de routage et les règles de pare-feu. Par défaut, un tel espace de

noms est incapable de communiquer avec l'extérieur à travers le protocole IP. Une manière de router des paquets réseau vers l'extérieur est de créer une paire d'interfaces réseau virtuelles, communément appelées *veth*. Un utilisateur peut créer une telle interface s'il est autorisé à effectuer des tâches d'administration dans chaque espace de noms, c'est-à-dire s'il possède la capacité *CAP_NET_ADMIN*. La création d'interfaces virtuelles permet de créer un pont entre deux espaces de noms différents.

Utilisateur

L'espace de noms utilisateur est apparu avec la version 3.8 de Linux en 2013. C'est le seul espace de noms utilisable par les utilisateurs non privilégiés. C'est aussi sans aucun doute l'espace de noms le plus complexe du fait de l'impact très large qu'il peut avoir sur le comportement du noyau, affectant sa gestion des utilisateurs, et notamment les vérifications d'autorisation qui y sont liées.

Par conséquent, il n'est pas étonnant qu'un certain nombre de vulnérabilités aient été trouvées durant les années qui ont suivi l'ajout de cette fonctionnalité. Ces vulnérabilités ont été corrigées au fur et à mesure de leurs publications. Par mesure de prévention, certaines distributions Linux comme Debian ont mis en place un garde-fou (*sysctl* `kernel.unprivileged_userns_clone`) que l'administrateur doit explicitement désactiver pour permettre l'utilisation des espaces de noms utilisateur par des utilisateurs non privilégiés. Par la suite, nous supposerons que cette fonctionnalité est suffisamment mature pour qu'elle soit accessible à des utilisateurs non privilégiés.

Chaque espace de noms utilisateur peut avoir une table de correspondance pour les identifiants utilisateur (UID) et les identifiants de groupe (GID). Un tel environnement permet aux utilisateurs de changer d'identifiant, et donc d'obtenir l'identifiant 0. Cet identifiant, qui représente l'utilisateur *root*, administrateur de la machine, ne permet toutefois pas d'obtenir tous les privilèges usuels de celui-ci. En effet, les actions d'un utilisateur *root* dans un espace de noms particulier ne peuvent impacter que les processus qui s'y trouvent également. Cependant, ces privilèges restreints permettent d'effectuer certaines tâches d'administration dans cet environnement particulier. Il est par exemple possible de créer d'autres espaces de noms pour les processus du même espace de noms utilisateur. La création d'un espace *mount* est ainsi permise, à quelques conditions près pour des raisons de sécurité. Typiquement, des fichiers originellement invisibles dans un espace de noms doivent le rester, quelles que soient les manipulations effectuées.

3.2.3 Mise en place d'un environnement pour les sujets

Pour ne pas avoir à modifier la TCB et être utilisable sur tous les systèmes basés sur Linux, *StemJail* tire parti des espaces de noms utilisateur. Ces derniers sont utilisés pour la création de cages. L'utilisateur *root* d'un espace de noms utilisateur peut créer ses propres espaces de noms *mount*. Il est ensuite possible d'utiliser des *bind-mounts* à l'intérieur de cet espace de noms pour exposer un sous-ensemble de fichiers et de dossiers choisis, avec les permissions correspondant aux options de montage : lecture seule ou lecture-écriture. L'environnement adéquat pour un sujet peut ainsi être créé. Quelques points techniques requièrent cependant une attention particulière, ce que nous détaillons dans les paragraphes suivants.

Premièrement, un point de montage récursif ne devrait pas être créé sans prendre en compte les points de montage hérités. Étant donné que les options de montage pour un *bind-mount* s'appliquent seulement au premier point de montage de la source, il est nécessaire que *StemJail* se charge des points de montage imbriqués pour éviter d'exposer des fichiers avec un accès inadapté dans une cage. Pour éviter ce comportement par défaut,

StemJail monte récursivement les chemins qui doivent légitimement être exposés. Le nombre de points de montage peut alors devenir important, ce qui dépend du nombre de points de montage présents dans l'espace de noms initiaux, c'est-à-dire celui de l'hôte, qui peut atteindre plus d'une vingtaine dans une distribution GNU/Linux standard. Pour tous les points de montage et les fichiers et dossiers exposés, *StemJail* doit dans un premier temps créer le point de montage, puis le remonter en lecture seule si nécessaire, et enfin faire de même pour tous les points de montage hérités. Pour éviter des *race-conditions*, ces ajustements sont dans un premier temps effectués dans un dossier temporaire avant que ses nouvelles ressources soient exposées de manière atomique aux sujets de la cage. La croissance du nombre de *bind-mounts*, due au remontage des arborescences de points de montage, ne pose pas de réel problème en pratique grâce à la faible empreinte mémoire d'un point de montage.

Dans un second temps, une bonne gestion des montages implique de garder le contrôle de leur hiérarchie. Typiquement, si `/a` est monté sur `/b`, il faut décider si de nouveaux points de montage dans `/a` sont propagés à `/b`. Pour ce faire, un point de montage peut être marqué comme partagé (*shared*), esclave (*slave*) ou privé (*private*). Si nous n'interdisons pas la propagation des nouveaux points de montage, il est trivial de contourner la politique de sécurité. Par exemple, il suffit de monter un périphérique de stockage USB sur `/media` alors que ce dossier est exposé en lecture seule dans une cage.

Par ailleurs, la plupart des applications nécessitent un accès à certains périphériques Linux pour fonctionner correctement. Par exemple, `/dev/null` et `/dev/urandom` sont largement utilisés. Ces périphériques sont des interfaces vers le noyau qui peuvent donner accès à des données potentiellement sensibles, ce qui est particulièrement le cas pour les périphériques de stockage et leurs partitions comme `/dev/sda1`. Être *root* dans un espace de noms utilisateur ne suffit pas pour créer de tels périphériques. Pour cette raison, *StemJail* monte certains périphériques parents de l'hôte dans chaque cage. Par mesure de défense en profondeur, nous avons choisi de n'exposer que quatre périphériques nécessaires : `null`, `full`, `zero` et `urandom`.

Le dossier temporaire `/tmp` est également largement utilisé par les applications. Ce dossier est traditionnellement partagé entre tous les utilisateurs du système pour stocker des fichiers temporaires. Pour éviter tout risque lié à une erreur d'utilisation de ce dossier qui pourrait aider un attaquant à acquérir plus de privilèges ou des accès imprévus, nous avons choisi de créer un dossier de stockage des fichiers temporaires par cage. Cette approche est plus prudente et ne change pas le comportement attendu des applications. De plus, ce dossier privé peut être utilisé pour stocker des données tout en étant certain que d'autres cages ne peuvent pas y accéder, ce qui permet par exemple de protéger les sockets UNIX s'y trouvant.

Les sujets peuvent avoir besoin de lister les processus actuellement en fonctionnement et d'interagir avec eux, par exemple via un signal. Il n'est cependant pas souhaitable qu'ils puissent interférer avec les processus s'exécutant en dehors de leur cage. Pour se protéger de cette menace, il est nécessaire de maîtriser l'exposition du système de fichiers *proc* habituellement monté dans le dossier `/proc`. Grâce à l'espace de noms PID, *StemJail* crée un pseudo système de fichiers dédié à chaque cage.

3.2.4 Fonctionnement interne de *StemJail*

Les cages sont séparées les unes des autres et du système hôte en utilisant les espaces de noms *mount*, UTS, PID et utilisateur. Chaque processus dans une cage est donc incapable de voir ou d'interagir avec des processus extérieurs à sa cage. La vue du système de fichiers peut cependant être partagée en accord avec les domaines définis par l'utilisateur.

Les canaux de communication entre les sujets et leur moniteur reposent sur des sockets

UNIX. Les sockets sont ouverts dans le dossier `/tmp`, qui est privé et dédié à une instance de cage. Cette architecture permet d'identifier de manière sûre l'origine d'une requête et donc de se protéger contre des attaques du type *confused deputy* [35].

Le système de fichiers attaché à une cage est construit de manière incrémentale. Quand une cage démarre, le flux d'exécution se déroule comme suit :

1. dans un premier temps, un moniteur, de PID 1 dans sa cage, crée une nouvelle racine de système de fichiers avec un contenu minimal : `/dev`, `/tmp` et `/proc`. Cette nouvelle hiérarchie de fichiers est la seule visible et accessible à tous les autres processus dans la cage. Le moniteur doit pouvoir l'étendre avec de nouvelles ressources, il garde donc accès au système de fichiers parent. Ce système de fichiers parent lui est exposé par un *bind-mount* récursif de la racine de l'hôte conservé dans un point de montage dédié. Ce répertoire devient le dossier courant du moniteur. Bien entendu, seul le moniteur (et non ses sujets) doit pouvoir accéder à ce point de montage : ce dernier est recouvert par un autre point de montage par le moniteur, ce qui a pour effet de le rendre inaccessible aux autres processus de la cage. En effet, Linux garantit que les sujets ne peuvent pas accéder aux ressources de leur moniteur, car contrairement à eux ce moniteur est privilégié dans cette cage ;
2. le moniteur expose ensuite dans la cage, via un *bind-mount* (en lecture seule), les fichiers exécutables et les ressources partagées fournies par le système. L'exécution de ce fichier continue ensuite normalement ;
3. tout au long de la vie d'une cage, chaque tentative inédite d'accès d'un sujet à une ressource déclenche une requête vers le moniteur. Ce dernier peut ensuite ajouter des fichiers dans la cage, en suivant la politique de sécurité, quand une requête est reçue et accordée. Si besoin, le moniteur effectue un *bind-mount* en lecture seule ou en lecture-écriture d'un ensemble de fichiers ou de dossiers à partir du système de fichiers parent. Le moniteur complète le système de fichiers de la cage pour qu'il corresponde au domaine atteignable le plus général. Tout processus qui essaye d'accéder à un fichier ou dossier ne le trouvera pas s'il n'a pas été précédemment monté.

La sécurité de la solution repose sur deux fondements. D'une part, les requêtes au moniteur doivent être le seul moyen pour un sujet d'obtenir de nouveaux accès. Autrement dit, le cloisonnement fourni par une cage doit être efficace. Cette propriété est garantie par le noyau Linux, incluant l'implémentation des espaces de noms, qui fait partie de la TCB. D'autre part, les sujets ne doivent pas être capables de compromettre le moniteur, qui représente le point central de la sécurité d'une cage.

3.2.5 Sûreté d'implémentation du moniteur

Afin de protéger le moniteur *StemJail*, il est souhaitable qu'il soit développé dans un langage sûr pour éviter les erreurs habituelles engendrant des failles de sécurité. Nous avons choisi Rust [3], un langage de programmation système conçu pour la performance tout en mettant à disposition des fonctionnalités de gestion sûre de la concurrence et de la mémoire. Comme l'expliquent Anderson *et al.* [36], le typage fort associé à la notion de *lifetime* ainsi que la programmation par *trait* apporte des outils puissants pour l'écriture de code robuste. Ces propriétés aident à prévenir un large éventail de vulnérabilités récurrentes comme les dépassements de tampon (*buffer overflow*), l'utilisation de ressources libérées (*use after free*) ou de pointeurs invalides (*dangling pointers*), ou encore l'utilisation de mémoire non initialisée.

Rust offre de plus des fonctionnalités bas niveau comme la manipulation directe de la mémoire ou des capacités d'interopérabilité avec des fonctions externes (*foreign function*

interface), ce que nous utilisons pour gérer des interfaces bas niveau comme les espaces de noms Linux, la manipulation des descripteurs de fichier ou encore la configuration de terminal virtuel. Les protocoles réseau utilisés par *StemJail* pour les communications entre les clients utilisateur, le portail et les moniteurs sont créés avec des automates finis (*finite-state machines*) et de la sérialisation de données. La modularité du compilateur Rust permet d'utiliser un système de *plugin* pour étendre automatiquement le code de *StemJail*, ce que nous utilisons pour générer automatiquement des fonctions de sérialisation sûres à partir des structures de données. De plus, nous utilisons le système de typage pour nous assurer à la compilation que seules des transitions valides sont possibles dans les automates de *StemJail* (*typestate analysis*).

3.2.6 Intégration transparente avec les applications

Quand un utilisateur ouvre une session sur le système, son *shell*, qu'il soit texte ou graphique, est configuré pour lancer automatiquement le portail *StemJail*. Pour s'intégrer aisément avec le *shell* utilisateur, la manière la plus simple est de remplacer les raccourcis des applications par des *wrappers*. Une approche alternative est de précharger une bibliothèque de *hooks* dans le *shell* utilisateur pour remplacer l'utilisation de *wrappers* sans nécessiter de modifications des raccourcis d'application. Le lanceur *StemJail*, invoqué par les *wrappers*, prend une application avec ses arguments en paramètre et demande la création d'une nouvelle cage au portail *StemJail*. De cette manière, chaque application lancée par l'utilisateur démarre son premier processus dans une cage. Utiliser *StemJail* de cette manière permet d'éliminer la charge cognitive liée au choix des rôles et des domaines. Si besoin, l'utilisateur est également libre de lancer manuellement *StemJail* pour un sujet particulier dans le domaine de son choix.

Pour s'intégrer dans un système existant, sans modifications statiques des applications, *StemJail* tire parti des fonctionnalités de *hooks* du *linker* dynamique. *StemJail* utilise le système *preload* pour charger des morceaux de son code dans les applications. Cette fonctionnalité de *preload* a pour but de remplacer les fonctions fournies par la bibliothèque C standard, dans notre cas la *libc*. Ceci s'applique aux exécutables binaires ELF liés dynamiquement qui sont donc amenés à utiliser les *wrappers* de la *libc* sur les appels système. Se reposer sur des exécutables liés dynamiquement nous semble acceptable étant donné que les distributions GNU/Linux fournissent très majoritairement des binaires ELF dynamiques.

Les garanties de sécurité de *StemJail* ne reposent évidemment pas sur le *wrapping* d'appels système. En effet, *StemJail* implémente le principe d'interdiction par défaut : seuls les fichiers qui ont été exposés explicitement par le moniteur d'une cage sont visibles aux sujets. C'est donc bien la robustesse des cages qui empêche des processus malveillants d'accéder à des fichiers de manière illicite. Placer des *hooks* sur les appels système fournit une solution pratique pour avoir des sujets compatibles avec le système de requêtes vers leur moniteur, mais n'est pas utilisé comme une fonction de sécurité.

Comme le montre la figure 3.2, le code chargé via la fonctionnalité de *preload*, appelé *shim*, est une bibliothèque partagée qui comprend la partie cliente de *StemJail* et un cache pour limiter le nombre de requêtes. Étant donné que le cache fait partie de la bibliothèque partagée dont le code est chargé et contrôlé par le processus sujet, aucune confiance ne peut lui être accordée. Le but du code de *preload* est de placer un *hook* pour chaque appel système lié au système de fichiers, par exemple *open* pour l'ouverture de fichier, *stat* pour la récupération de métadonnées ou encore *rmdir* pour la suppression de dossiers. Quand un sujet effectue un appel système via le *shim*, ce dernier commence par effectuer une recherche dans son cache pour déterminer si une requête est nécessaire. Quand une requête est jugée nécessaire, le *shim* établit une connexion avec le moniteur à travers un

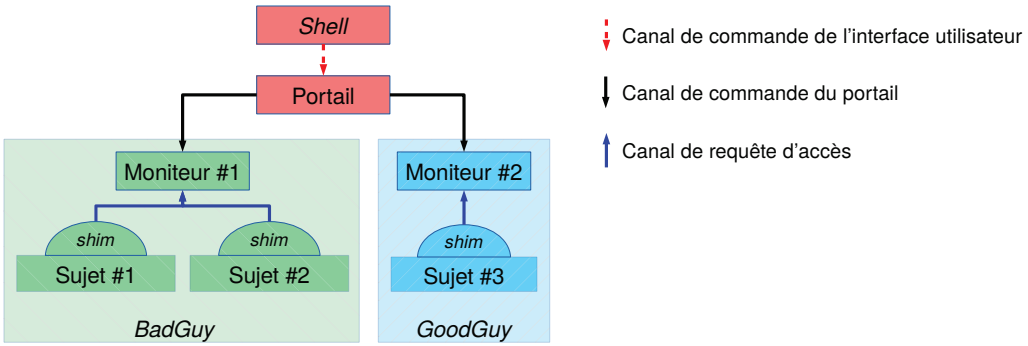


FIGURE 3.2 – Détails d'une instanciation de *StemJail*

socket UNIX uniquement accessible dans la cage du sujet et envoie la demande d'accès. Le moniteur peut ensuite soit évoluer vers un nouveau domaine et effectuer un *bind-mount* des fichiers appropriés, soit rester dans le domaine courant lorsqu'aucun domaine atteignable n'autoriserait l'accès. Lorsqu'une évolution est effectuée par le moniteur, il renvoie la liste des nouveaux fichiers et dossiers accessibles afin de raffiner le cache du sujet. Le code du *hook* de l'appel système se termine ensuite et passe la main à l'exécution de l'appel système. L'appel système peut réussir, si le fichier ou le dossier est présent avec des droits appropriés, ou échouer avec une erreur provenant du noyau qui indique que le fichier ou le dossier n'existe pas. Ainsi, des appels système effectués par un processus malveillant ne peuvent être fructueux que lorsqu'ils impliquent une ressource visible dans la cage.

Il est important de souligner que le moniteur ne prend pas en compte le cache des clients, mais analyse chaque requête pour juger si elle est légitime ou non. Les caches sont utilisés pour limiter le nombre de requêtes émises afin d'améliorer les performances. Étant donné que le moniteur ne monte pas une ressource en fonction de sa présence dans le cache du client, l'utilisation d'un tel cache ne peut pas avoir d'impact sur la sécurité de *StemJail*.

Les logiciels existants listent couramment le contenu des dossiers, ce que *StemJail* doit permettre pour une intégration transparente. Cela implique de laisser les processus récupérer la liste des fichiers potentiellement accessibles, et seulement ceux-là. En conséquence, ces informations peuvent représenter une fuite de métadonnées sur les fichiers qui peuvent se révéler inaccessibles par la suite à cause d'un changement de domaine. Nous estimons cependant que cela ne remet pas en cause la politique de sécurité de l'utilisateur, qui est principalement centrée sur le contenu des fichiers. Voici l'exemple d'un tel cas. Comme illustré sur la figure 2.1 (page 23), nous considérons une cage dans laquelle le moniteur est dans le domaine courant `bg`, et qu'un sujet demande à lister le dossier `~/`. Le dossier `~/Clients/` est visible et devrait donc apparaître. Les fichiers `~/Clients/BadGuy/` et `~/Clients/GoodGuy/` sont accessibles à travers des transitions de domaine distinctes, de telle manière que le dossier `~/Clients/` devrait les lister tous les deux. Cependant, `~/Accounts/` ne devrait pas être listé étant donné qu'il n'est pas actuellement visible et ne peut pas devenir accessible via une transition de domaine.

La fonctionnalité de *preload* du *linker* dynamique peut être utilisée soit avec une variable d'environnement nommée `LD_PRELOAD`, avec le fichier `ld.so.preload` ou encore avec le fichier `ld.so.cache` du dossier `/etc`. Pour ce qui est de la variable d'environnement, son affectation ne nécessite aucun privilège et est héritée d'une exécution à l'autre dans la plupart des cas. L'inconvénient de cette technique est que l'environnement peut facilement être réinitialisé par une application. Au contraire, le fichier `ld.so.preload` est situé dans le dossier `/etc`, qui est habituellement la propriété de l'administrateur et donc non modifiable par d'autres utilisateurs. Cependant, *StemJail* crée chaque partie du système de fichiers d'une cage et peut donc également créer ce fichier. De plus, utiliser ce fichier est plus pratique que

d'utiliser `ld.so.cache`, étant donné que ce dernier peut être écrasé à la suite d'une mise à jour du système. En pratique il y a plus d'une quarantaine d'appels de fonctions qui doivent avoir un *hook*, chacune avec ses propres arguments. Chaque fonction a été dûment associée à une commande de requête cliente.

L'utilisation de la fonctionnalité de *preload* a un impact négligeable sur les performances. Le code du client fait partie de chaque processus ; chaque client peut calculer le nombre minimal de requêtes à émettre vers le moniteur tout en maintenant un cache des requêtes précédentes et de leurs réponses. Ce système permet au moniteur de ne recevoir qu'un faible nombre de requêtes, pertinentes pour le sujet qui les émet. De plus, il n'y a pas de pénalité due à un changement de contexte entre un sujet et son moniteur contrairement aux autres méthodes de *hook* comme *ptrace*.

3.2.7 Interaction utilisateur

Installation et configuration

Les binaires de *StemJail* sont soit installés par l'administrateur pour tout le système, soit par un utilisateur pour sa propre utilisation. Les espaces de noms utilisateur doivent être activés dans le noyau. Placé à un niveau d'abstraction au-dessus des espaces de noms Linux, *StemJail* est indépendant de l'architecture matérielle. La configuration est séparée de l'installation de telle manière que n'importe quel utilisateur peut utiliser *StemJail* comme il le souhaite.

La configuration, qui inclut la définition du rôle et de l'activité utilisateur, est prise en compte pour un domaine quand il est créé et jusqu'à la fin de sa vie. Toute modification de la configuration est ignorée par les domaines instanciés pour éviter des états incohérents du moniteur.

StemJail est construit pour l'utilisateur final et doit donc être configurable simplement. Chaque domaine est décrit dans un fichier dédié en suivant le format TOML. Ce format clef-valeur est simple et adapté pour une prise en main facile par un utilisateur. La figure 3.3 présente un exemple de configuration pour le profil *GoodGuy*. Les sections `[[fs.bind]]` permettent de grouper des autorisations d'accès. Chacune de ces sections prend une clef `path` et un chemin de fichier ou de dossier comme valeur. Par défaut, l'accès à un ficher décrit par un tel chemin n'est autorisé qu'en lecture seule. L'utilisateur peut activer un accès en lecture-écriture en ajoutant une clef `write` avec la valeur `true`. Une section optionnelle `[run]` peut prendre une commande d'exécution par défaut. Enfin, le nom du domaine est renseigné par la valeur de la clef globale `name`.

Interfaces utilisateur

StemJail supporte l'usage de deux types d'interfaces utilisateur : l'interface en ligne de commande et l'interface graphique.

Interfaces en ligne de commande (*command-line interfaces* : CLI) Un périphérique de terminal (TTY) ou pseudo-terminal Linux (PTY) est composé d'une extrémité maître et d'une autre esclave. L'extrémité maître représente le terminal utilisateur qui peut envoyer des commandes comme des touches pressées, le redimensionnement de la fenêtre ou encore des interruptions. En retour, cette extrémité peut recevoir du texte envoyé par les processus qui ont accès à l'extrémité esclave, comme c'est typiquement le cas d'un *shell* texte. Comme illustré sur la figure 3.2 (page 39), le portail de *StemJail* est utilisé pour transférer les interactions utilisateur aux sujets via leur moniteur de cage. Le portail crée une nouvelle instance de pseudo-terminal et expose l'extrémité esclave dans la cage alors que l'extrémité maître est transférée à son client sur le terminal utilisateur.

```
# profile to manage the GoodGuy client

name = "GoodGuy"

[[fs.bind]]
path = "/bin"

[[fs.bind]]
path = "/etc"

[[fs.bind]]
path = "/lib"

[[fs.bind]]
path = "/lib64"

[[fs.bind]]
path = "/usr"

[[fs.bind]]
path = "/var/cache"

[[fs.bind]]
path = "/home/user/Clients/GoodGuy"
write = true

[run]
cmd = ["/bin/sh"]
```

FIGURE 3.3 – Exemple de fichier de configuration pour le profil *GoodGuy*

L'implémentation de ce proxy de pseudo-terminal pour *StemJail* tire avantage de l'appel système *splice* pour effectuer des transferts zéro-copie entre les extrémités maître et esclave. Cette optimisation réduit significativement l'impact sur les performances dû au traitement des échanges.

Interfaces graphiques Afin de protéger le serveur d'affichage de techniques d'exploitation comme la capture d'écran, le *keylogger* ou encore l'espionnage du presse-papiers [37, 38], les applications graphiques ne devraient pas y avoir directement accès. La sécurité du protocole *X Window System* ne prend en compte que l'authentification des clients, qui se connectent potentiellement à travers le réseau. Ce protocole n'a pas été conçu pour protéger l'utilisateur de clients authentifiés (mais non de confiance), comme c'est le cas d'un logiciel malveillant s'exécutant dans la session utilisateur. Un proxy X comme Xbox [39] ou un bureau virtuel transparent comme Xpra permettent de créer un pare-feu de commandes graphiques entre une cage et le serveur d'affichage de l'utilisateur. Les problématiques liées à la sécurité de l'interface homme-machine seront détaillées dans le chapitre 5.

3.3 Évaluation

3.3.1 Scénario d'utilisation

Nous poursuivons notre exemple pour illustrer l'impact de *StemJail* sur différentes activités. Nous esquissons un *workflow* simple et classique : l'utilisateur *Bob* lance un

explorateur de fichiers et navigue jusqu'au fichier de comptes du client *BadGuy*, sur lequel il veut travailler. Il ouvre alors ce fichier avec l'application de tableur *OfficeSheet*. Par la suite, *OfficeSheet* se fait compromettre par le document en question et un processus malveillant essaye alors d'exfiltrer des données du système. Les traces 3.1 à 3.5 montrent des extraits de journaux générés par des composants de *StemJail* pendant ce scénario.

```
1  Loaded configuration: profiles: ["BadGuy", "GoodGuy", "MyBank"]
2  Portal got request: Run(DoRun(RunRequest { profile: None, command: ["/usr/bin/file-explorer"] }))
3  Running jail: BadGuy || GoodGuy || MyBank
4  Child jailing
5  Creating tmpfs in /proc/fs/nfsd
6    Bind mounting from /usr to /proc/fs/nfsd/usr
7    [...]
8  Creating tmpfs in /tmp
9  Populating /dev
10   Creating tmpfs in /dev
11   Bind mounting from /dev/null to /proc/fs/nfsd/dev/null
12   [...]
13   Creating tmpfs in /dev/shm
14 Pivot root
15 Got jail PID: 2
16 Waiting for child 2518 to terminate
```

TRACE 3.1 – Initialisation de la cage

La trace 3.1 est générée lorsque l'utilisateur lance l'explorateur de fichiers à travers un lanceur *StemJail*, comme décrit dans la section 3.2.6. Le lanceur se connecte au démon du portail et demande la création d'une cage. Le portail démarre alors une instance de moniteur avec comme état initial le domaine le plus général, à savoir `bgm`, qui correspond au rôle issu de la disjonction *BadGuy ou GoodGuy ou MyBank* (ligne 3). Ce moniteur crée ensuite un dossier qui lui servira de dossier de travail (*working directory*) pour préparer le système de fichiers initialement exposé dans la cage (lignes 5 à 13). Cette hiérarchie est construite dans un système de fichiers temporaire de type `tmpfs`, indépendant des systèmes de fichiers hôte. Le moniteur peuple ensuite le système de fichiers de la cage avec les fichiers nécessaires (`/tmp`, `/dev`...) tel que décrit dans la section 3.2.4. Finalement, l'explorateur de fichiers exécute un processus, correspondant à *OfficeSheet*, dans cette nouvelle cage (lignes 15 à 16). L'utilisateur est maintenant capable d'accéder aux documents de tous les domaines à travers une transition de domaine.

```
1  Monitor got request: Shim(Access(AccessRequest { data: AccessData { path: "/home", write: false }
       ↪ }))
2    No domain reachable: access denied
3  Monitor got request: Shim(Access(AccessRequest { data: AccessData { path: "/home/user", write:
       ↪ false } }))
4    No domain reachable: access denied
```

TRACE 3.2 – Requêtes d'accès refusées avant la compromission du processus

La trace 3.2 montre les requêtes d'accès que le moniteur n'accorde pas. Ici le moniteur ne monte pas les dossiers `/home` (lignes 1 à 2) ou `/home/user` (lignes 3 à 4) car il n'y a pas de domaines, parmi ceux définis, qui autorisent un tel accès. Si le moniteur effectuait le montage, les processus de la cage auraient accès à toute leur arborescence, ce qui inclut les fichiers qui sont supposés n'être accessibles par aucun domaine. En ce qui concerne le dossier racine, le sujet peut bien entendu utiliser son propre dossier racine, mais les éléments

présents dans ce dernier ne sont pas les mêmes que ceux qui apparaissent à la racine du système de fichiers parent.

```
1 Monitor got request: Shim(Access(AccessRequest { data: AccessData { path: "/home/user/Clients/
      ↪ BadGuy", write: false } }))
2   Bind mounting from ./parent/home/user/Clients/BadGuy to ./tmp_mount_zLpqpb2ojEGn/
3   Moving bind mount from ./tmp_mount_zLpqpb2ojEGn to /home/user/Clients/BadGuy
4   Removed ./tmp_mount_zLpqpb2ojEGn
5   Domain transition: BadGuy || GoodGuy || MyBank -> BadGuy
6   Access granted to [AccessData { path: "/home/user/Clients/BadGuy", write: true }]
```

TRACE 3.3 – Transition de domaine

La trace 3.3 est générée par l'utilisateur qui navigue dans les dossiers pour atteindre le document qui se trouve dans `~/Clients/BadGuy/`. L'application *OfficeSheet* envoie une demande d'accès pour ce chemin au moniteur. Cette fois-ci, il existe un domaine, *BadGuy*, atteignable à partir du domaine courant du moniteur, dans lequel ces accès sont listés. De plus, étant donné que cet accès est seulement autorisé à partir de ce domaine particulier, il s'agit bien du domaine le plus général qui autorise cet accès. Le moniteur déclenche donc une transition vers le domaine *BadGuy* et effectue un *bind-mount* de tous les nouveaux chemins rendus disponibles, qui incluent au moins le chemin demandé, depuis le système de fichiers parent vers celui de la cage (ligne 2). Les opérations de montage sont dans un premier temps effectuées dans un dossier temporaire, ce qui nous permet d'éviter une *race condition*. Par la suite, ces points de montage sont déplacés de manière atomique dans la partie de la cage visible par les sujets (ligne 3).

```
1 Monitor got request: Shim(Access(AccessRequest { data: AccessData { path: "/home/user/Clients/
      ↪ BadGuy/.", write: false } }))
2   Current domain already allows this access
3   Access granted to []
4 Monitor got request: Shim(Access(AccessRequest { data: AccessData { path: "/home/user/Clients/
      ↪ BadGuy/malicious.xlsm", write: false } }))
5   Current domain already allows this access
6   Access granted to []
```

TRACE 3.4 – Requêtes d'accès autorisées

La trace 3.4 montre des demandes d'accès légitimes déjà autorisées par le domaine *BadGuy* (lignes 1 et 4). Chaque *thread* des processus dispose de son propre cache, ce qui nécessite tout de même de remplir initialement ce cache par les premières requêtes d'accès. Nous constatons qu'un sujet, correspondant à l'instance d'application *OfficeSheet*, demande l'accès à un fichier de tableur.

```
1 Monitor got request: Shim(Access(AccessRequest { data: AccessData { path: "/home/user/Clients/.",
      ↪ write: false } }))
2   No domain reachable: access denied
3 Monitor got request: Shim(Access(AccessRequest { data: AccessData { path: "/home/user/Clients/
      ↪ GoodGuy", write: false } }))
4   No domain reachable: access denied
```

TRACE 3.5 – Requêtes d'accès générées suite à la compromission du processus

Il est important de rappeler que *StemJail* ne détecte pas les processus malveillants, mais traite tous les processus comme potentiellement malveillants. La trace 3.5 illustre

ce qui se passe après que l'instance d'application *OfficeSheet* a été compromise par un fichier malveillant dans le domaine *BadGuy*. Le processus compromis peut alors accéder aux données de *BadGuy*, pour les exfiltrer ou les chiffrer dans le cas d'un rançongiciel (*ransomware*). Les journaux montrent la demande du sujet malveillant pour obtenir l'accès aux ressources de *GoodGuy*, via une requête cliente *StemJail* (lignes 1 et 3). Étant donné qu'il n'est pas possible de transiter vers un domaine qui autoriserait ces *bind-mounts*, le moniteur décline la demande. Pour rappel, le processus malveillant ne gagnerait rien en effectuant un appel système sans passer par le moniteur : les fichiers ne lui sont pas accessibles dans la cage où il s'exécute.

Le moniteur s'arrête lorsque tous ses sujets terminent ou quand l'utilisateur demande son arrêt explicite via une requête au portail. L'utilisateur peut en effet mettre un terme à tous les processus d'une cage lorsqu'il le souhaite, ce qui peut être particulièrement utile si des processus d'une cage abusent des ressources du système, comme les processeurs.

3.3.2 Revue des erreurs communes de sécurité pour un contrôle d'accès en espace utilisateur

Comme expliqué par Garfinkel [30], certaines failles de sécurité typiques sont couramment trouvées dans les mises en œuvre de *sandbox* : *la duplication incorrecte de l'état du système d'exploitation*, *la réplication incorrecte du code du système d'exploitation* et *des effets de bord inattendus liés aux refus d'appels système*. Dans les paragraphes suivants, nous survolons les erreurs usuelles, en expliquant comment *StemJail* s'en prémunit.

Contrairement à ce qui est classiquement fait dans des logiciels de *sandboxing*, la sécurité de *StemJail* ne repose pas sur le filtrage des appels système ou la modification de leurs arguments basée sur des décisions prises tout en maintenant une copie de l'état du système d'exploitation. En effet, notre implémentation n'interdit pas les appels système et ne modifie pas leurs arguments. *StemJail* a comme effet de bord d'étendre la vue du système de fichiers des sujets en fonction de leurs requêtes. Comme la représentation du système de fichiers de la cage et les domaines atteignables constituent les seules informations pertinentes à notre application de la politique, il en résulte un état assez simple à maintenir dans le moniteur. De plus, le moniteur est le seul processus qui modifie la vue du système de fichiers, et il le fait de manière atomique, comme décrit dans la section 3.2.3. Enfin, le moniteur est la seule partie du code responsable de l'application de sa politique. Premièrement, toutes les ressources visibles dans une cage ont été montées par le moniteur via des appels système qu'il a lui-même effectués. Deuxièmement, le moniteur n'émule pas le noyau. Le seul comportement du noyau sur lequel la bonne application de la politique repose est le fait qu'une ressource non visible dans la cage ne doit pas y être accessible.

Une erreur similaire consiste à *répliquer de manière incorrecte le code du système d'exploitation*, dans notre cas du noyau. Si jamais *StemJail* réplique incorrectement ce code, par exemple pour la mise sous forme canonique de chemin de fichier, ceci ne doit pas mener à des problèmes de sécurité car la même sémantique est utilisée pour effectuer les demandes des nouveaux accès au noyau. La mise sous forme canonique de chemin est faite au niveau du sujet, mais vérifiée par le moniteur pour éviter des interprétations incohérentes, comme cela pourrait être le cas avec des chemins relatifs. Le problème principal résiduel pourrait être une incompréhension de la requête du sujet, mais la politique de sécurité sera respectée de toute manière. Par exemple, si des liens symboliques n'étaient pas correctement gérés par le moniteur, le pire cas de figure serait de transiter vers un domaine non souhaité par le sujet. Cependant, ceci ne représenterait qu'un problème fonctionnel : la transition devrait de toute façon être autorisée par la politique de l'utilisateur. La politique de sécurité, dont la mise à jour est gérée par le moniteur, est appliquée par le noyau.

Un autre type d'erreur peut venir de la *mauvaise interprétation des chemins indirects vers*

des ressources. Ceci est utilisé pour créer des canaux de communication non souhaités entre sujets supposément restreints par une politique de sécurité. *StemJail* résout indirectement ce type de problèmes en garantissant que le système de fichiers créé par le moniteur est la seule ressource partagée entre les cages. Il en résulte que les ressources partagées (du système de fichiers) auxquelles les sujets peuvent accéder dépendent de la politique de l'utilisateur exprimée pour le domaine. Comme vu dans la section 3.2.1, le processus initial d'une cage est le moniteur qui lance ensuite le premier sujet. Ce deuxième processus est exécuté sans ressources partagées autres que le système de fichiers créé par le moniteur. Les ressources partagées que les sujets peuvent acquérir dépendent de la politique de l'utilisateur. Par exemple, lors de la définition de la politique, il faut prendre en compte la présence de sockets UNIX qui pourraient être partagées entre des domaines qui ne doivent pas être capables de coopérer. En effet, Linux autorise les connexions à ce type de sockets même si le système de fichiers est en lecture seule.

Les vulnérabilités basées sur une incohérence entre le moment de vérification d'un accès et le moment d'utilisation de cet accès (*time of check to time of use* : TOCTOU [40]) forment un autre type de problèmes de sécurité. L'architecture du moniteur de *StemJail* est multitâche, ce qui est notamment utile pour la gestion concurrentielle de plusieurs sujets, propriété importante pour de bonnes performances. Pour autant, notre utilisation des propriétés de concurrence fournies par le langage Rust permet de garantir que les transitions de domaine effectuées par le moniteur sont séquentielles, ce qui est nécessaire pour exclure toute *race condition* dans l'interprétation de la politique de sécurité. Les autres *race conditions* comme le déréférencement d'arguments ne sont pas pertinentes dans le cas de *StemJail* étant donné qu'il n'y a ni interposition d'appels système ni manipulation des requêtes d'accès, qui sont par ailleurs entièrement copiées vers le moniteur. Un appel système retournera naturellement une erreur si aucune requête auprès du moniteur n'a précédemment rendu la ressource visible dans la cage.

3.3.3 Impact sur les performances

Pour évaluer les performances de *StemJail*, nous présentons dans la table 3.1 des tests comparables à ceux utilisés par Potter *et al.* [41] ainsi que par Kim *et al.* [26]. La méthodologie utilisée est détaillée dans un script fourni avec le code source de *StemJail*[1]. Le système utilisé pour les expérimentations comprend un processeur Intel Xeon 2GHz utilisant 2 cœurs avec l'*hyper-threading* activé (c.-à-d. 4 cœurs logiques), 4GB de RAM (DDR2, 667 MHz), avec un système d'exploitation Debian Linux et un noyau Linux 4.4. Pour chaque test, nous fournissons des moyennes de temps d'exécution, avec un système de fichiers stocké sur un disque dur mécanique (colonne HDD), et avec un système de fichiers stocké en RAM via un point de montage *ramfs* (colonne RAM). La colonne HDD est nécessaire pour pouvoir comparer nos tests sur *StemJail* avec d'autres tests de performance, et la colonne RAM est intéressante pour mesurer l'impact sur les tests liés à la lenteur d'un disque mécanique.

Les résultats sont cohérents avec le comportement attendu de *StemJail*. La surcharge la plus importante causée par *StemJail* est concentrée sur le cache des sujets, ce qui diminue significativement l'impact sur les performances.

Le test Gunzip décompresse le fichier d'archive (`tar.gz`) correspondant à la version 4.4 de Linux. Ce test stresse le système de fichiers par des opérations de lecture de l'archive et des opérations d'écriture vers le fichier extrait. Excepté pour les premiers accès aux fichiers, notamment l'archive, il n'y a par la suite plus de communications avec le moniteur, donc plus de pénalité visible.

1. `https://github.com/stemjail/stemjail/blob/master/tools/bench.sh`

	Natif		*StemJail*			
Test	**HDD**	**RAM**	**HDD**		**RAM**	
Gunzip	16,04s	6,13s	16,03s	0,0%	6,13s	0,0%
Untar	68,30s	1,57s	69,95s	2,4%	1,97s	25,4%
Zip	36,36s	31,76s	38,42s	5,6%	33,49s	5,4%
Build-1	1137,38s	1134,10s	1190,16s	4,6%	1188,02s	4,7%
Build-4	320,96s	315,09s	344,69s	7,3%	330,37	4,8%

TABLE 3.1 – Résultats de tests de performance (*StemJail* 0.4.0)

Le test Untar consiste à extraire l'archive (`tar`) décompressée. Ce test montre un faible impact sur les performances, sauf lorsque lancé sur un système de fichiers *ramfs*. En effet, extraire un grand nombre de fichiers sur le système de fichiers nécessite un usage intensif du cache du sujet. Cependant, la charge apportée par *StemJail* est minimale vis-à-vis des opérations d'accès aux données dans le cas d'un disque mécanique, et est au contraire prépondérante dans le cas de l'utilisation d'un système de fichiers entièrement en mémoire vive.

Le test Zip crée une nouvelle archive (`zip`) à partir des fichiers provenant de l'archive originale. Ce test implique un grand nombre d'accès au système de fichiers comme pour Untar, mais nécessite le parcours et la lecture de tous les fichiers sources. Les calculs importants requis par l'opération de compression, absents avec Untar, forment le goulot d'étranglement qui masque les opérations de lecture du système de fichiers, même pour un *ramfs*.

Le test Build-1 consiste à compiler la version 4.4 de Linux sur un seul cœur logique du processeur (1 *job*). Ce test crée un grand nombre de processus à faible durée de vie qui accèdent à un grand nombre de fichiers différents. Nous utilisons la configuration par défaut pour l'architecture *x86_64*. Les résultats montrent un impact minimal lors d'une compilation dans une cage. Le test Build-4 est également une compilation à partir des mêmes fichiers, mais en utilisant les 4 cœurs logiques du processeur (4 *jobs*). Ceci a pour conséquence de créer un grand nombre d'accès concurrents au système de fichiers, ce qui peut mettre beaucoup de tension sur un moniteur qui reçoit une requête par tentative d'accès. Cependant, grâce à un cache efficace dans chaque processus, le moniteur *StemJail* reçoit relativement peu de requêtes et l'impact sur les performances reste constant.

Bien que non précisés par les auteurs, les tests de performance du *sandboxing* de Mbox [26] basé sur *ptrace* et optimisé avec *seccomp-bpf* semblent avoir été effectués sur un disque mécanique. En comparaison, *StemJail* est trois à cinq fois plus performant. Ce résultat confirme l'intérêt de notre intégration d'un cache au sein de chaque sujet afin d'éviter des transactions inutiles.

3.3.4 Comparaison avec d'autres solutions

Dans le contexte d'une distribution GNU/Linux ordinaire reposant sur du contrôle d'accès discrétionnaire, l'isolation de processus appartenant à différentes activités nécessite la création d'un compte utilisateur par activité. L'utilisateur final pourrait alors utiliser chacun de ces comptes pour passer d'une activité à l'autre et les groupes utilisateur pour partager des données entre elles. Bien entendu, cette approche est peu pratique et la gestion de comptes utilisateur nécessite des droits d'administration. Dans le cadre de notre exemple présenté dans la section 2.1 (page 18), il ne serait pas possible de créer un compte

utilisateur dédié pour un nouveau client de *Bob* sans octroyer à ce dernier, directement ou indirectement, des droits d'administration. Un autre point bloquant est l'absence de prise en compte de changements de rôles. L'utilisateur doit manuellement changer de compte sans qu'il y ait de possibilité de découverte automatique d'activité. Par ailleurs, peu d'isolation est présente entre chaque utilisateur, ce qui permet notamment de voir les commandes effectuées par les autres utilisateurs.

Dans le contexte d'un système GNU/Linux utilisant SELinux, notre but consisterait à créer de multiples rôles accessibles par un utilisateur. Chaque rôle devrait être dédié à un client comme *BadGuy* et associé à une application comme *OfficeSheet*, utilisées comme points d'entrée, pour faciliter des transitions entre rôles. Cependant, l'ajout des nouveaux clients de *Bob* demanderait la création de nouveaux rôles, ce qui nécessite des droits d'administration de SELinux. De plus, une transition vers un domaine ne peut pas être déclenchée dynamiquement lors de l'accès à un fichier. Ceci empêche une adaptation dynamique de politique comme décrite dans le chapitre 2. Les mêmes inconvénients peuvent être soulevés pour les autres modules de sécurité Linux tels qu'AppArmor, Smack ou Tomoyo qui sont également conçus pour être configurés par un administrateur de la machine, ce qui ne correspond pas à notre besoin.

Le même constat est valable pour les solutions de cloisonnement telles que LXC, Docker, Linux-VServer ou OpenVZ. Même si certaines d'entre elles utilisent les espaces de noms utilisateur, elles requièrent toutes un processus administrateur pour gérer un conteneur, que ce soit un service lancé ou un binaire privilégié. En effet, les espaces de noms utilisateur ne fournissent pas toutes les fonctionnalités requises à l'administration des conteneurs, par exemple la création de fichiers de périphérique ou encore la configuration de ponts réseau. A contrario, *StemJail* n'offre pas toutes ces fonctionnalités, mais a été conçu pour tirer pleinement partis des espaces de noms utilisateur, utilisables par tout utilisateur non privilégié.

StemJail utilise une bibliothèque partagée (*shim*) pour envoyer des requêtes à un processus externe de confiance, mais n'émule pas les appels système pour le contrôle d'accès. Le seul recours possible à l'émulation a lieu pour l'action de lister le contenu des dossiers, ce qui n'est pas directement lié au contrôle d'accès. De plus, le *shim StemJail* utilise un cache pour n'envoyer des requêtes que lorsque strictement nécessaire, ce qui diminue grandement la charge supplémentaire. Contrairement à Mbox, nous n'utilisons pas `ptrace`, ce qui permet ainsi de s'affranchir des problèmes liés aux nombreux changements de contexte avec le moniteur, ce qui offre de meilleures performances.

Bien que certains projets orientés utilisateur existent, à notre connaissance, aucun ne fournit une fonctionnalité de cloisonnement dynamique et automatique comme le permet *StemJail*. De plus, contrairement à *StemJail*, peu de solutions de virtualisation ou de cloisonnement sont développées avec un langage fortement typé qui apporte de bonnes propriétés de gestion de la mémoire, comme le fait Rust. Ces propriétés sont cependant fortement souhaitables pour avoir confiance dans une solution de sécurité.

3.3.5 Limitations et évolutions possibles de *StemJail*

L'approche que nous avons suivie avec *StemJail* apporte des avantages certains par rapport aux autres solutions de cloisonnement. Cependant, ce projet a mis en évidence certaines limitations techniques liées aux fonctionnalités du noyau Linux et à l'espace utilisateur qui ne sont pas à négliger.

StemJail ne traite actuellement que des chemins de fichiers. La prise en compte du réseau est également une extension intéressante pour une politique de sécurité. En effet, ceci pourrait permettre de prolonger le contrôle d'accès à des ressources accessibles en dehors de celles propres au système d'exploitation. Notre volonté de ne pas nécessiter de privilèges

est cependant difficilement conciliable avec l'usage de l'espace de noms réseau. En effet, comme expliqué dans la section 3.2.2, les espaces de noms utilisateur ne sont pas autorisés à interférer avec un espace de noms parent, comme l'espace de noms réseau. Par exemple, il est interdit de créer une interface réseau de type *veth* pour router le trafic réseau entre deux espaces de noms réseau, et donc d'y appliquer des règles de filtrage via la configuration d'un pare-feu. Une solution possible serait d'utiliser un proxy réseau applicatif, par exemple un proxy *socks*, éventuellement avec un support transparent comme *tproxy* fourni par Netfilter. Cette solution n'est cependant pas entièrement satisfaisante en termes de flexibilité, de protocole accessible et de performance. Une fonctionnalité Linux permettant nativement de faire du contrôle d'accès, sans nécessiter d'être privilégié, pourrait repousser cette limite des espaces de noms.

Enfin, dans l'architecture actuelle s'appuyant sur les mécanismes actuellement disponibles dans le noyau Linux, le moniteur partage des espaces de noms avec les processus qu'il supervise. Des fonctionnalités supplémentaires de contrôle d'accès fournies par le noyau pourraient apporter une isolation plus importante des composants critiques de *StemJail*.

3.4 Conclusion

Nous avons décrit dans ce chapitre le projet libre *StemJail*, une implémentation du contrôle d'accès présenté dans le chapitre 2. Ce mécanisme permet d'appliquer automatiquement la séparation des informations conformément à une configuration effectuée par l'utilisateur. Nous revenons en conclusion sur nos objectifs établis dans la section 3.1.

Le premier objectif était de mettre en œuvre un contrôle d'accès adapté à l'utilisateur. La découverte automatique de rôle permet de déduire progressivement l'activité d'un utilisateur et de créer des cages adaptées. L'implémentation de *StemJail* est possible grâce à l'utilisation d'espaces de noms de Linux qui permettent de créer des cages isolées. Un moniteur est chargé de faire évoluer une cage en fonction du comportement des processus qui y sont présents et de la politique de sécurité définie par l'utilisateur.

Le deuxième objectif était de satisfaire les besoins de tous les utilisateurs d'un système GNU/Linux standard, pour qu'ils soient en mesure de se protéger et d'atténuer les conséquences de l'exécution d'un logiciel malveillant. Pour cela, il fallait que *StemJail* ne requière aucun privilège d'administration. Cet objectif est atteint grâce à l'utilisation des espaces de noms utilisateur.

Aussi, pour pouvoir s'adresser à un utilisateur non expert, la définition des politiques doit être suffisamment simple pour qu'il puisse la créer et la comprendre. Ce troisième objectif est atteint grâce à notre utilisation de l'organisation des fichiers en dossiers.

Afin de ne pas dégrader la sécurité du système, suite à une escalade de privilèges qui pourrait être introduite par une faille dans *StemJail*, aucune modification de la TCB du système n'est requise. Ce quatrième objectif est atteint par l'utilisation de bonnes pratiques de développement associées à un langage de programmation approprié comme Rust.

Le cinquième objectif était de ne pas nécessiter de modifications des applications pour permettre une adoption aisée. La compatibilité applicative est assurée par la surcharge dynamique de bibliothèques du système, sans que ceci puisse impacter la sécurité de notre solution.

Enfin, le sixième objectif était d'avoir un impact minimal sur les performances lors de l'utilisation de *StemJail*. Ceci est atteint grâce à des composants logiciels *multithread* et un système de cache efficace pour chaque processus cloisonné.

Chapitre 4

Landlock : cloisonnement programmable non privilégié

Sommaire

Ce chapitre traite de *Landlock*, une nouvelle solution open-source [1] de contrôle d'accès qui peut être utilisée sans privilèges d'administration. L'objectif est de permettre à un processus de se placer dans une *sandboxe*, c'est-à-dire de définir et de s'appliquer des règles de contrôle d'accès afin de limiter l'impact de sa compromission éventuelle ainsi que de celle des processus qu'il a engendrés. *Landlock* permet de remédier au manque de granularité des espaces de noms Linux, notamment pour améliorer les cages de *StemJail*.

La section 4.1 justifie l'intérêt et décrit les différents objectifs de *Landlock*. La section 4.2 présente les principales propriétés de *Landlock*, notamment la notion de contrôle d'accès programmable. La section 4.3 détaille l'implémentation actuelle de *Landlock* pour Linux ainsi que la création, les fonctionnalités et les contraintes des règles de contrôle d'accès. Enfin, la section 4.4 permet d'évaluer *Landlock* en termes de performance, de surface d'attaque et d'utilisation.

4.1 Objectif

Pour la première implémentation du modèle développé dans le chapitre 2, *StemJail* exploite les primitives actuellement mises à disposition par le noyau Linux. Malgré la flexibilité des espaces de noms Linux, la section 3.3.5 a souligné les limitations de leur usage pour le contrôle d'accès. Une première limitation concerne le manque de finesse de définition d'un contrôle d'accès. En effet, il n'est pas possible d'appliquer une politique de sécurité sur certains objets noyau ou sur certaines actions. Par exemple, les transferts réseau ou l'utilisation des descripteurs de fichiers ne peuvent pas être finement contrôlés. Il n'est pas non plus possible d'autoriser la lecture du contenu d'un répertoire tout en restreignant l'accès à son contenu. Au vu de ces limitations, nous pouvons nous interroger sur la pertinence de l'utilisation des fonctionnalités actuellement offertes par le noyau Linux pour appliquer un contrôle d'accès complet mis en place par un utilisateur non privilégié. Il faudrait qu'un utilisateur non privilégié soit capable de s'appliquer des contraintes par un moniteur qui lui serait dédié. Ce moniteur doit avoir suffisamment de privilèges pour manipuler tous les objets d'intérêt et être capable de restreindre finement leur accès. Il ne doit cependant pas être capable d'abuser de cette position pour élever les privilèges de l'utilisateur. Aucun des mécanismes existants du noyau Linux ne permet de répondre à ce besoin.

Ce chapitre introduit *Landlock*, un nouveau mécanisme permettant d'effectuer du contrôle d'accès non privilégié sans les limitations des espaces de noms Linux. À l'instar de SELinux, *Landlock* est un module de sécurité Linux qui partage les mêmes points de contrôle du noyau mais dispose de sa propre logique. La comparaison s'arrête là, car le but est bien que l'utilisateur définisse et mette en place ses propres règles de contrôle d'accès, sans intervention de l'administrateur.

Se pose alors la question de la forme que peuvent prendre ces règles. Le chapitre précédent identifiait clairement la contrainte de ne pas modifier la TCB, dont le noyau fait intégralement partie. Même si un compromis est à trouver dans le présent chapitre, nous nous montrons conservateurs : nous choisissons d'utiliser les programmes eBPF, un mécanisme déjà implémenté et largement utilisé, pour décrire des règles de contrôle d'accès. De la même manière que l'utilisation des espaces de noms Linux n'est pas triviale, nous supposons que l'écriture de ces programmes est laissée à la discrétion d'un développeur expert. Comme évoqué précédemment, nous souhaitons être capables de retranscrire le contrôle d'accès souhaité par l'utilisateur. Nous considérons donc que les règles *Landlock* peuvent être présentées à l'utilisateur avec un langage de plus haut niveau comme le fait *StemJail* avec ses fichiers de configuration.

1. `https://landlock.io` (cf. huitième version de la série de patchs)

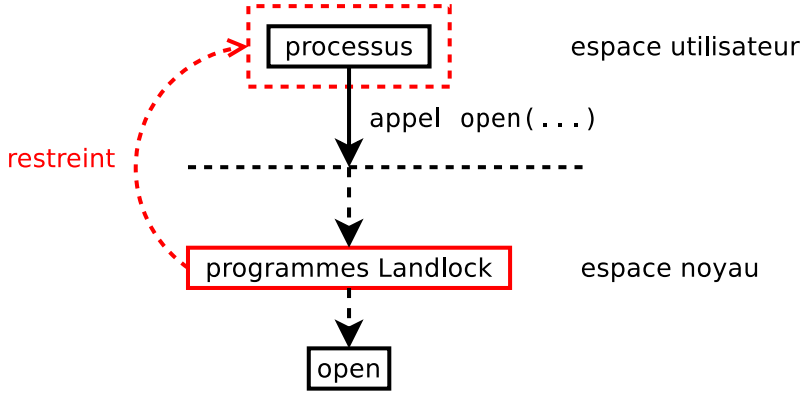


FIGURE 4.1 – Vue macroscopique du contrôle d'accès d'une ouverture de fichier

La figure 4.1 donne une idée du fonctionnement attendu. Les actions d'un processus sont validées par un ensemble de programmes *Landlock* attachés à ce processus. Cette figure illustre le contrôle d'accès appliqué à l'ouverture d'un fichier.

4.2 Expressivité et propriétés du contrôle d'accès mis en place par *Landlock*

Landlock apporte des concepts et une base de travail pour implémenter de multiples types de contrôle d'accès. Bien qu'il soit extensible, le contrôle d'accès détaillé dans ce chapitre ne tient compte que du système de fichiers.

4.2.1 Contrôle d'accès programmable

Le contrôle d'accès programmable permet d'inclure des règles de contrôle d'accès dans des applications. Les développeurs de ces applications ont ainsi la possibilité de définir une politique de sécurité adaptée à leur cas d'usage. La politique embarquée peut permettre de créer et d'utiliser des rôles adaptés pour chacun des composants d'une application, indépendamment de la politique du système sur lequel l'application est exécutée. Cette approche permet d'appliquer le principe de défense en profondeur par le raffinement du contrôle d'accès.

L'expressivité d'un contrôle d'accès programmable est fonction du langage de programmation utilisé. Il est donc possible d'implémenter un contrôle d'accès obligatoire, discrétionnaire ou encore basé sur des rôles comme présenté dans la section 1.1. *Landlock* est conçu pour permettre à chaque utilisateur, privilégié ou non, de restreindre davantage la politique de sécurité qui lui est imposée, en y ajoutant son propre contrôle d'accès via ses applications. Trois concepts permettent d'exprimer un contrôle d'accès avec *Landlock* : les permissions, les programmes et les règles.

Permission Une permission *Landlock* correspond à un type d'action sur un objet. Concernant les fichiers, les permissions disponibles sont similaires à celles de SELinux : lecture, écriture, exécution, ajout, création, verrouillage...

Programme Un programme *Landlock* est une suite d'instructions interprétée par le noyau. Un tel programme est responsable de valider les demandes d'accès correspondant à un ensemble de permissions donné. Il prend en entrée une référence à un objet et retourne

en sortie une autorisation ou un refus. Par la suite le terme « programme » désignera un programme *Landlock*.

Règle Un programme peut être chaîné à un autre pour former une séquence de programmes capable de raffiner l'identification d'un objet. Une règle *Landlock* correspond à une telle séquence appliquée sur un *sujet, correspondant ici à un ensemble de processus*. Cette séquence est interprétée lorsqu'une action du sujet sur un objet nécessite une permission *Landlock*. Un exemple de règle sera détaillé dans la section 4.3.5.

4.2.2 Mise à jour et verrouillage du contrôle d'accès

Un système de contrôle d'accès dynamique comme *StemJail* doit être capable de définir et de faire évoluer un environnement cloisonné. Pour ce faire, *Landlock* offre d'une part la possibilité de stocker des références à des objets noyau, et d'autre part un mécanisme de mise à jour de ces ensembles de références. Les mises à jour sont possibles par les programmes *Landlock* mais également par les processus à leur origine, ce qui sera détaillé dans la section 4.3.6.

Landlock n'autorise que l'ajout de nouvelles restrictions, ce qui rend impossible l'acquisition de nouveaux accès. Une fois les programmes *Landlock* assignés et verrouillés, il n'est plus possible de désactiver leur application. À l'inverse, il est possible d'augmenter à la volée les restrictions d'accès sur un processus en empilant de nouveaux programmes *Landlock*.

4.2.3 Application de règles *Landlock* sur des sujets

Un sujet *Landlock* est défini comme un ensemble de processus. L'application d'une règle sur un sujet s'effectue en deux temps. La première étape consiste à créer et charger un ensemble de programmes *Landlock* dans le noyau via l'appel système `bpf`. Un programme *Landlock* est référencé par un descripteur de fichier en espace utilisateur. Ce descripteur peut ensuite être utilisé pour appliquer le programme sur un processus.

L'application d'un programme *Landlock* sur un processus donné se fait par lui-même. Un sujet *Landlock* évolue en incluant tous ses futurs processus fils. L'application de ce programme s'étendra donc à tous les futurs fils de ce processus à la manière des filtres *seccomp-bpf*. Pour ce faire, l'appel système `seccomp` est utilisé avec une commande dédiée à *Landlock* et avec le descripteur de fichier d'un programme. La bonne prise en compte d'un programme *Landlock* peut être validée par le code retour de l'appel système. Si l'appel système réussit, les décisions de contrôle d'accès prises par ce programme seront alors intégrées à la chaîne de contrôle d'accès du système pour les processus supervisés.

Cette approche par supervision de la hiérarchie d'un processus est particulièrement adaptée au durcissement d'un logiciel via la création de *sandbox*. Le développeur du logiciel intègre à son initialisation la création d'un environnement cloisonné. La politique de sécurité est définie par le développeur pour son logiciel et est appliquée en complément de la politique du système.

4.2.4 Composition de règles *Landlock* et intégration avec les autres contrôles d'accès

Il existe de nombreux modèles de contrôle d'accès et de langages pour les définir. Les approches usuelles s'appuient sur un rôle administrateur pour l'application de la politique de sécurité du système. Il en résulte un point unique de définition et d'agrégation des politiques de sécurité. *Landlock* propose une approche de composition de politiques de

contrôle d'accès [42, 43] qui permet d'appliquer différentes politiques simultanément. Il faut pour cela gérer les conflits lors de l'application et de l'interprétation des règles.

La composition de règles *Landlock* avec le contrôle mis en place par l'administrateur est également un prérequis pour un système de contrôle d'accès accessible à des utilisateurs non privilégiés. En effet, des règles appliquées sur un même processus peuvent avoir des objectifs différents. Les actions d'un processus doivent être autorisées par la liste de règles *Landlock* qui lui est potentiellement appliquée, mais l'autorisation initiale doit être préalablement accordée par les autres systèmes de contrôle d'accès du noyau. L'ordre d'interprétation des programmes *Landlock* correspond à l'ordre inverse de leur application, comme c'est le cas pour *seccomp-bpf*. Pour qu'une action soit autorisée par *Landlock*, tous les programmes conçus pour valider cette action doivent l'autoriser. Après l'interprétation du programme le plus récemment ajouté à la liste attachée à un processus, chacun des autres programmes de cette liste est interprété tant que son prédécesseur autorise la requête.

Les politiques *Landlock* sont définies par des ensembles de règles qui sont également composables avec les autres mécanismes de contrôle d'accès du noyau. L'autorisation d'accès finale correspond à la validation de cet accès par tous les autres systèmes de contrôle du noyau, ainsi que de l'autorisation de toutes les règles *Landlock* appliquées au processus pour une action donnée.

4.2.5 Exemple récapitulatif

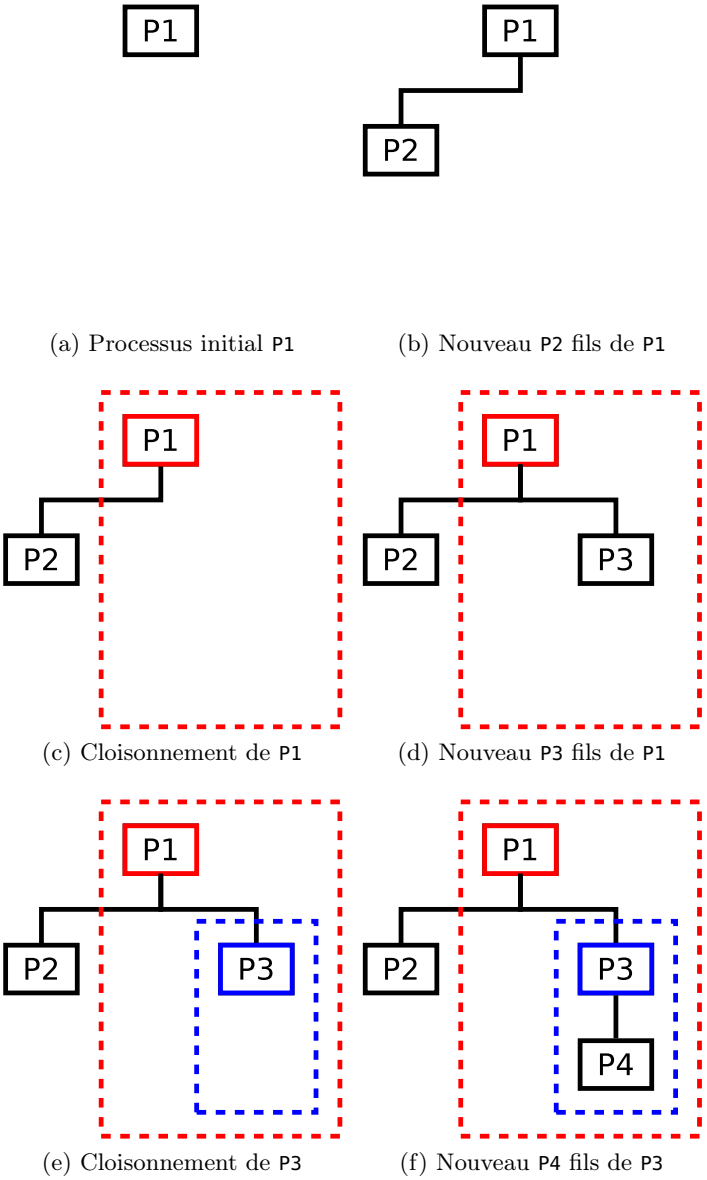
La figure 4.2 présente une arborescence de plusieurs processus soumis à des programmes *Landlock*. Dans cet exemple, le processus `P1` crée un processus fils `P2` (sous-figure 4.2b). Aucun de ces processus n'est soumis au contrôle d'accès *Landlock*. Dans un second temps, le processus `P1` s'applique un cloisonnement avec un ensemble de programmes *Landlock* (sous-figure 4.2c). À partir de ce moment, tous les processus qui seront créés par `P1` ou sa future descendance se verront également appliquer les mêmes programmes. C'est ce qui se produit lorsque le processus `P3` est créé (sous-figure 4.2d). Lorsqu'un processus souhaite s'appliquer un cloisonnement, alors les règles *Landlock* sont composées. Quand le processus `P3` est soumis à un nouveau cloisonnement (sous-figure 4.2e), il conserve tout de même celui mis en place par le processus `P1`, que celui-ci soit toujours en vie ou non. Les contraintes mises en place par `P1` et `P3` s'appliquent automatiquement aux nouveaux processus engendrés par `P3`, ce qui inclut le processus `P4` (sous-figure 4.2f). Si une vulnérabilité est exploitée sur le processus `P4`, l'attaquant ne pourra alors accéder qu'aux ressources autorisées par les règles mises en place par `P1` et `P3`. Les contraintes de sécurité portant sur l'application de règles sur un processus seront détaillées dans la section 4.4.2.

4.3 Implémentation des règles et infrastructure dédiée à leur application

Les règles *Landlock* sont des ensembles de programmes écrits par un développeur d'application avec l'aide d'outils. Cette section explique le fonctionnement des différents mécanismes du noyau Linux utilisés et adaptés pour *Landlock* concernant la création, la vérification et l'utilisation de ces programmes dans l'infrastructure de sécurité Linux.

4.3.1 Programme eBPF

Un langage est nécessaire pour écrire une règle de contrôle d'accès. Il peut y avoir différents niveaux d'abstraction pour exprimer un contrôle d'accès, du haut niveau pour l'utilisateur au bas niveau pour le noyau. Un langage de haut niveau pourrait correspondre



(a) Processus initial `P1` (b) Nouveau `P2` fils de `P1`

(c) Cloisonnement de `P1` (d) Nouveau `P3` fils de `P1`

(e) Cloisonnement de `P3` (f) Nouveau `P4` fils de `P3`

FIGURE 4.2 – Évolution de l'application de programmes *Landlock* sur une hiérarchie de processus

à celui de *StemJail*, présenté dans la section 3.2.7. Nous nous intéressons ici à la partie bas niveau, équivalente à de l'assembleur, qui sera directement interprétée par le noyau.

Ce langage doit à la fois être suffisamment flexible et offrir de bonnes garanties de sécurité pour son utilisation par le noyau. En effet le *parsing* et l'interprétation de données sont souvent à l'origine de failles de sécurité. L'interprétation d'une règle ne doit menacer ni l'intégrité du noyau, ni la politique de sécurité du système. L'infrastructure *extended Berkeley Packet Filter* (eBPF) apporte des garanties nécessaires pour se prémunir contre ces risques. Il s'agit d'une version étendue de *classic Berkeley Packet Filter* [44] (BPF ou cBPF) initialement créée pour permettre le filtrage efficace des paquets réseau. Un programme *Landlock* est un programme eBPF conçu spécifiquement pour une règle *Landlock*. Les données fournies à des programmes *Landlock* ne sont pas des paquets réseau, mais un contexte mettant à disposition les données nécessaires à une prise de décision. Un exemple

de contexte sera détaillé dans la section 4.3.5.

Les programmes eBPF sont une suite d'instructions (*bytecode*) interprétées par une machine virtuelle située dans le noyau Linux. Ce *bytecode* a été conçu pour être compatible avec celui de cBPF, mais également pour se traduire facilement dans le jeu d'instructions d'un processeur 64-bits. Ceci permet une optimisation et surtout une transformation efficace pour une machine de cette architecture matérielle. Nous pouvons remarquer que l'interpréteur cBPF est un simple traducteur vers du code eBPF pour les architectures usuelles. Les composants qui utilisent cBPF, voire un sous-ensemble de ces instructions comme le fait *seccomp-bpf*, profitent donc également de la machine virtuelle eBPF.

4.3.2 Vérification de programme eBPF

Les programmes eBPF proviennent de l'espace utilisateur et doivent donc être considérés comme potentiellement malveillants ou contenant des bogues. Lors du chargement d'un programme eBPF dans le noyau par un processus, ce dernier transmet le *bytecode* eBPF ainsi que des métadonnées qui indiquent notamment l'usage souhaité du programme. Il existe différents types de programmes eBPF en fonction de leur utilisation, ce qui est également lié au niveau de privilège nécessaire pour les charger dans le noyau. Le type d'un programme est précisé lors de son chargement dans le noyau. Un type définit les données qui seront accessibles au programme, les moyens permettant d'exploiter ces données, ainsi que le code noyau qui peut faire appel au programme. Par exemple, il existe des types de programmes pour le filtrage de paquets réseau, pour le routage ou encore pour le débogage. Suivant son type, un programme peut apporter différentes fonctionnalités.

Lors d'une demande de chargement d'un programme dans le noyau, le programme est soumis à une analyse statique qui permet de valider un certain nombre de propriétés importantes développées ci-après. Il n'existe pas encore de formalisation ni de preuve du vérificateur, mais un travail important des développeurs du noyau Linux a permis de valider empiriquement le fonctionnement attendu, notamment grâce aux nombreux tests présents dans les sources de Linux.

Limitation du nombre d'instructions

Tout d'abord, il faut être capable de limiter la taille d'un programme eBPF afin de ne pas saturer la mémoire du noyau. La taille maximale d'un programme est aujourd'hui de 4096 instructions. De plus, la taille de la mémoire occupée par tous les programmes d'un utilisateur est imputée sur la taille de la mémoire qu'il est autorisé à verrouiller.

Validité des instructions

L'étape suivante de vérification consiste à valider que le *bytecode* ne contient que des instructions valides. Ceci permet de s'assurer qu'il n'y aura pas d'erreurs d'interprétation du *bytecode*, ce qui pourrait entraîner un comportement inattendu.

Exécution en temps fini

Garantir une exécution en temps fini permet de se protéger contre des attaques en saturation des ressources du noyau. La limite du nombre maximal d'instructions d'un programme eBPF est complétée par la contrainte d'avoir uniquement pour graphe de flot de contrôle des graphes acycliques directs. Cette restriction est garantie par le fait qu'il n'existe pas d'instruction pour faire un saut qui ne serait pas vers l'avant. Il n'est donc pas possible d'effectuer des boucles avec un programme eBPF, ce qui le rend par la même

occasion non Turing-complet. Cette contrainte limite l'expressivité d'un programme, mais le nombre d'instructions autorisées semble suffisant pour la majorité des cas d'usage légitimes.

Terminaison des branches d'exécution

Grâce à une émulation du programme, le noyau vérifie également que toutes les instructions sont atteignables. Il vérifie également que tout programme se termine bien par un code retour. Le code mort détectable par une analyse statique est par ailleurs rejeté.

Utilisation de la mémoire

La machine virtuelle comprend 11 registres qui sont lisibles et inscriptibles sous certaines conditions. Un registre n'est lisible que s'il a été précédemment initialisé par le même flux d'exécution. Ceci permet de protéger le noyau contre des fuites d'informations qui seraient présentes dans ces registres. Les données inscriptibles dans un registre peuvent entre autres venir de la pile du programme ou de son contexte.

La pile d'un programme eBPF est un espace mémoire qui lui est dédié pour stocker des valeurs durant son exécution. De la même manière que pour la lecture de registres, la lecture d'une valeur de la pile n'est autorisée que lorsque celle-ci a été précédemment initialisée.

Le contexte d'un programme eBPF a des propriétés similaires à la pile, mais contient des données qui proviennent directement du noyau. Ces données peuvent représenter le contenu d'un paquet réseau dans le cas d'un programme chargé de filtrer ce type de données, ou des données d'un appel système dans le cas d'un programme de type *seccomp-bpf*. L'écriture dans cet espace mémoire peut être autorisée ou non suivant le type de contexte et donc suivant le type de programme.

Typage fort des données

Lors du chargement d'une donnée dans un registre, le type associé à la donnée est transféré au registre pour l'analyse statique. Nous profitons ainsi des bonnes propriétés du typage fort, qui permettent d'éviter des classes de bogues, mais également d'étendre la validation statique d'un programme. Le typage est important pour avoir une interface saine vis-à-vis du noyau. Nous pouvons ainsi éviter de mauvaises interprétations de données qui pourraient mener à des erreurs, potentiellement exploitables.

L'utilisation des registres pour écrire dans la pile ou le contexte est également sujette au transfert de type. Nous sommes ainsi capable de valider la cohérence des types de données stockées et manipulées tout au long de la vie d'un programme.

Un aspect important de la vérification de type, concernant un registre ou une zone mémoire, est la cohérence de la taille des données impactées. La taille d'un registre est de 64 bits, mais il peut également stocker des données de taille inférieure. Il est donc nécessaire de vérifier la taille des données source et destination pour avoir une copie cohérente de la variable et ne pas faire fuiter une partie des données. Ceci permet notamment d'éviter des attaques de confusion de type qui consistent à invalider des hypothèses sur le contenu et le format des données, ce qui peut mener à des déréférencements arbitraires de pointeur ou encore à de l'exécution de code arbitraire.

Fuite d'adresses noyau

Le typage fort des données permet d'identifier des adresses noyau qui peuvent légitimement être accessibles à un programme eBPF, caractéristique nécessaire pour des

performances optimales. Dans le cas d'un programme non privilégié, il est cependant important d'interdire toute fuite d'information qui permettrait de faciliter une attaque. Les pointeurs noyau sont une source d'information intéressante pour un attaquant. Cela peut permettre d'identifier des zones mémoires utilisées par le noyau, ce qui est une information suffisante pour identifier l'organisation globale de la mémoire du noyau afin de l'exploiter. Pour éviter toute fuite de pointeurs, le vérificateur de programme eBPF valide le fait que les registres contenant une adresse ne peuvent être que copiés en gardant leur type, ou passés à une fonction. Ceci interdit donc toute opération arithmétique ou de test de valeur qui permettrait de faire fuiter indirectement la valeur d'une adresse.

4.3.3 Appliquer un programme *Landlock*

Infrastructure *Linux Security Module*

Linux Security Module [45] (LSM) est une infrastructure pour la création de composants de sécurité noyau. Un *hook* LSM correspond à un *Policy Decision Point* [46] (PDP), c'est-à-dire à une fonction qui a pour but d'évaluer le droit d'effectuer une action sur un type de ressource du noyau. Il existe plus de deux centaines de hooks utilisables en fonction de la configuration du noyau. Les appels de ces *hooks*, correspondant à des *Policy Enforcement Points* [46] (PEP), sont répartis dans le code du noyau aux emplacements pertinents pour des prises de décision atomiques concernant la sécurité du système. Ils précèdent l'exécution effective des requêtes de l'espace utilisateur sur les objets noyau qui lui sont visibles, comme les fichiers, les processus ou encore les sockets. Lors d'un appel système qui nécessite d'accéder à plusieurs ressources, le flot d'exécution du noyau peut donc passer par plusieurs PEP. Contrairement à *seccomp-bpf* qui se place au niveau des appels système, les *hooks* LSM ont à leur disposition la sémantique complète des objets noyau. Par exemple, lors de l'ouverture d'un fichier, le premier argument de l'appel système `open` contient un pointeur vers une chaîne de caractères censée identifier un chemin de fichier. Un filtre *seccomp-bpf* n'est ni capable d'accéder à cette chaîne de caractères, ni aux propriétés du fichier ainsi identifié. En effet, pour appliquer une politique de sécurité prenant en compte ces objets, l'interface des appels système est inadaptée. Utiliser les hooks LSM offre le bon niveau d'abstraction.

Dans la version 4.9 de Linux, il existe 8 composants qui utilisent cette infrastructure : le gestionnaire de capacités *POSIX.1e*, Yama, LoadPin, IMA/EVM, AppArmor, SELinux, Smack et Tomoyo. Chacun d'eux implémente un ensemble de *hooks* LSM qui permettent d'appliquer une politique de sécurité particulière. L'infrastructure LSM permet ainsi de composer plusieurs composants de sécurité en fonction de la configuration souhaitée par l'administrateur du système.

Type de programme et hook *Landlock*

Landlock est un module de sécurité qui abstrait des hooks LSM avec des hooks *Landlock*, chargés de valider des actions sur un même type d'objets. Par exemple, les implémentations des hooks LSM `file_open` et `file_permission` font appel au hook *Landlock* `fs_pick` qui regroupe les opérations sur des fichiers. Pour la gestion des accès liés au système de fichiers, 33 hooks LSM sont utilisés par *Landlock*. Ils sont appelés lors d'un événement lié à une opération particulière comme l'ouverture de fichier, la création d'un nouveau fichier ou encore le renommage d'un fichier.

Un programme *Landlock* est un programme eBPF uniquement utilisable par le LSM *Landlock*. Il peut être interprété une ou plusieurs fois par un même hook *Landlock*. Il faut donc être capable d'associer un programme *Landlock* à un hook *Landlock*.

```
const union bpf_prog_subtype metadata = {
        .landlock_hook = {
                .type = LANDLOCK_HOOK_FS_PICK,
                .triggers = LANDLOCK_TRIGGER_FS_PICK_WRITE,
                .options = LANDLOCK_OPTION_PREVIOUS,
                .previous = prog_walk_fd,
        }
};
```

FIGURE 4.3 – Exemple de définition de métadonnées d'un programme *Landlock*

Nous avons ajouté une notion de sous-type aux programmes eBPF pour pouvoir attacher différentes métadonnées à un programme de type *Landlock*. La figure 4.3 est un exemple de définition de métadonnées qui doivent être fournies lors du chargement du programme eBPF correspondant à un programme *Landlock* dédié au contrôle d'accès sur le système de fichiers. Le champ `type` précise le hook *Landlock* auquel est destiné un programme *Landlock*. Le *bitfield* `triggers` indique le type d'action pour lequel le programme sera interprété et donnera une décision d'accès. Les champs `options` et `previous` sont détaillés par la suite.

Exemple d'application de contrôle d'accès

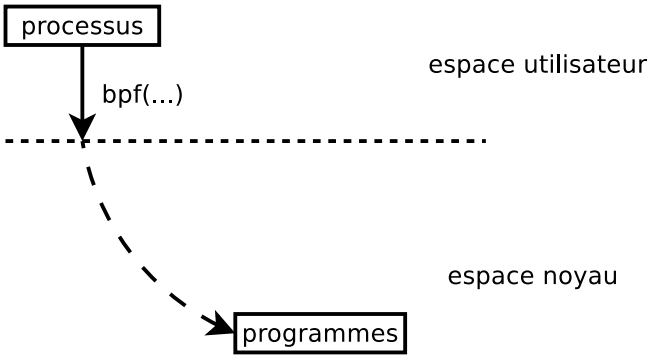
La figure 4.4 expose les étapes de la mise en place d'un *sandboxing*. Dans un premier temps le processus qui souhaite se placer dans une *sandbox* charge un ensemble de programmes *Landlock* dans le noyau via l'appel système `bpf` (sous-figure 4.4a). Ensuite, le processus s'applique des contraintes en s'attachant des programmes via l'appel système `seccomp` (sous-figure 4.4b). À partir de ce moment, les demandes d'accès à des ressources mises à disposition par le noyau peuvent être soumises à validation des programmes appliqués sur le processus. Dans le cas d'un accès à un fichier via l'appel système `open` (sous-figure 4.4c), le flot d'exécution du noyau passe par un ensemble de hooks LSM qui, après l'accord du contrôle d'accès traditionnel, peuvent appeler des hooks *Landlock*. Les programmes *Landlock* s'appliquant au processus émetteur de la requête ont alors la possibilité de refuser la demande. Si la demande est acceptée par tous les programmes, alors l'ouverture du fichier est finalisée.

4.3.4 Code noyau utilisable par un programme *Landlock*

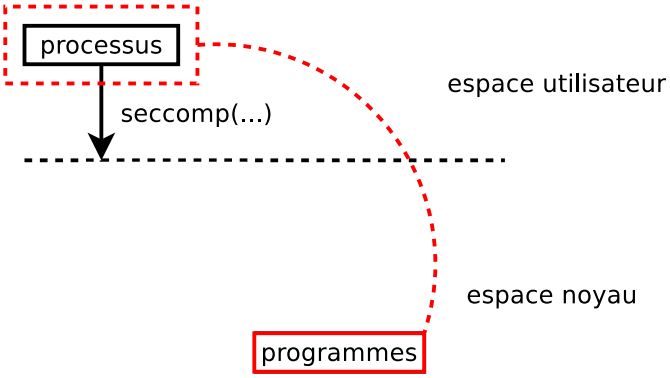
Un programme *Landlock* doit être capable de lire et de comparer des attributs des objets. Une référence à l'objet concerné par une action est mise à disposition des programmes dans leur contexte. La récupération des propriétés de cet objet passe par l'utilisation de fonctions.

Fonctions eBPF

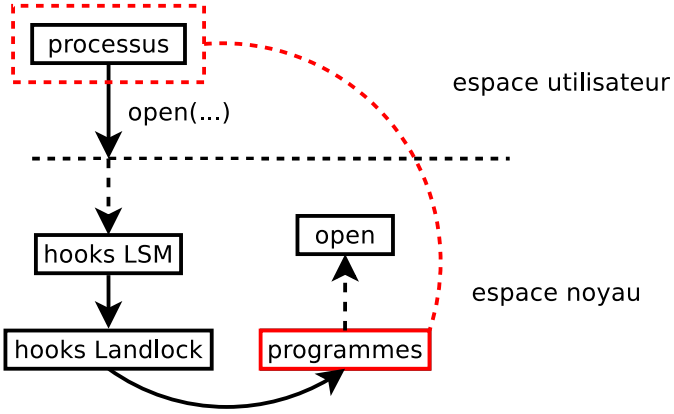
Contrairement aux programmes cBPF, chaque type de programme eBPF peut faire appel à des fonctions mises à disposition par le noyau. Cette interface, comparable à celle des appels système dédiée à l'espace utilisateur, permet à un programme eBPF d'appeler du code noyau dédié. Les fonctionnalités de tels programmes ne sont donc limitées que par le code fourni par ces fonctions. Le vérificateur de programme eBPF est chargé de valider les appels de fonctions des programmes. Si un programme fait appel à une fonction non autorisée pour son type de programme eBPF, alors le chargement de ce programme est refusé.



(a) Chargement de programmes *Landlock*



(b) Application de programmes *Landlock*



(c) Validation d'action par les programmes *Landlock*

FIGURE 4.4 – Étapes de *sandboxing* d'un processus

Les fonctions eBPF sont du code noyau statique écrit spécialement pour être appelé par un type de programme eBPF particulier, potentiellement malveillant. L'écriture de telles fonctions doit donc être effectuée avec soin pour ne pas remettre en cause les principes de sécurité de l'infrastructure eBPF. Le typage fort des arguments de fonction apporte des garanties importantes qui facilitent la validation de leur utilisation.

Fonctions accessibles aux programmes *Landlock*

Pour *Landlock*, ces fonctions eBPF sont une interface nécessaire pour permettre d'effectuer des opérations complexes qui nécessitent d'accéder à la représentation globale du système propre au noyau. Grâce aux types de programme eBPF clairement définis, il est possible d'avoir une liste blanche de fonctions eBPF accessibles aux programmes *Landlock*. Pour les types de programme *fs_ walk* et *fs_ pick*, nous avons implémenté et autorisé les appels des fonctions eBPF `bpf_inode_map_lookup` et `bpf_inode_get_tag`.

Enchaînement de programmes

Il est possible de chaîner plusieurs programmes *Landlock* interprétés au cours d'un même appel système. Un tel chaînage permet le partage d'un état entre une séquence de programmes, ce qui permet de créer une règle *Landlock* correspondant à une machine à états. Lorsque le champ `options` présent dans la structure de la figure 4.3 contient le drapeau `LANDLOCK_OPTION_PREVIOUS`, alors le champ `previous` est pris en compte. Ce champ contient un descripteur de fichier d'un programme *Landlock* précédemment chargé. Cette référence permet d'indiquer le programme à l'origine de l'état d'une règle. Suivant l'objet à valider, le noyau peut interpréter une séquence de programmes ainsi définie. Le noyau est donc garant de la terminaison d'une règle. Si l'option `LANDLOCK_OPTION_PREVIOUS` n'est pas présente, alors le programme se chaîne avec lui-même. La section suivante illustre de tels chaînages.

4.3.5 Exemple de règle pour le système de fichiers

Chaînage de programmes

Lors de l'accès à un fichier, que ce soit pour son ouverture ou toute autre action utilisant un chemin, au moins deux types de programmes *Landlock* sont interprétés séquentiellement. Ce découpage permet de raffiner l'identification d'un fichier au fur et à mesure du parcours d'une arborescence. Un premier type de programme est interprété pour chacun de ses dossiers parents et un deuxième type de programme permet de valider l'action finale demandée sur le fichier cible.

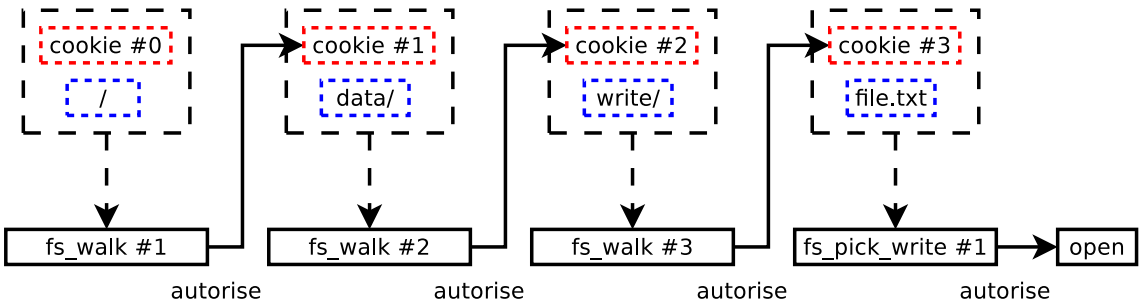


FIGURE 4.5 – Exemple de séquence de programmes `fs_walk` et `fs_pick_write`

La figure 4.5 correspond à l'interprétation d'une règle permettant de limiter l'accès d'une application présente dans une *sandbox* à un ensemble de hiérarchies de fichiers accessibles seulement en lecture. Cet exemple évalue l'accès au fichier `/data/write/file.txt` avec les programmes `fs_walk` et `fs_pick_write`. Un programme *Landlock* de type *fs_ walk* est dédié au parcours d'une arborescence de dossiers et est interprété pour chacun d'eux, sauf s'il s'agit du dernier élément d'un chemin. Un tel programme a la possibilité d'accorder l'accès ou non au dossier en question et donc au reste de l'arborescence. Dans cet exemple, un sujet effectue

une demande d'accès à un fichier. Le programme `fs_walk` est chargé de vérifier si ce fichier est contenu dans une liste blanche de dossiers. Ce programme est appelé pour les dossiers `/`, `data/` et `write/`. Chaque instance de programme inscrit son état dans la variable `cookie` d'une zone mémoire appelée contexte. À la prochaine itération, ce contexte contiendra également une référence au dossier ou fichier suivant. Un programme *Landlock* de type *fs_pick* est dédié à la validation d'une action sur un fichier donné, que ce soit une ouverture, une exécution, la récupération de ses métadonnées ou encore l'héritage d'un descripteur de fichier. Après le dernier dossier de l'arborescence, le programme `fs_pick_write` est interprété une fois pour le dernier élément du chemin : le fichier `file.txt`. S'il accorde la requête, alors l'appel système continue son exécution. Cette séquence permet donc de raffiner l'identification du fichier en validant chaque élément de sa hiérarchie.

Validation d'un chemin de fichier

Les programmes *Landlock* sont des programmes eBPF qui peuvent être générés à partir d'une fonction par le compilateur Clang. La syntaxe d'un programme s'apparente au langage C, mais seul un sous-ensemble des fonctionnalités de ce langage est supporté. Ceci permet d'être compatible avec les contraintes détaillées dans la section 4.3.2.

Les figures 4.6 et 4.8 représentent le contenu des programmes `fs_walk` et `fs_pick_write`. Pour chaque requête de demande d'accès, ces programmes sont interprétés et leur code de retour permet de refuser ou non l'accès. L'argument `ctx` fait référence au contexte de l'interprétation courante. La variable globale `dirs` fait référence à une liste de dossiers préalablement initialisée par le processus créateur du programme. Cette liste permet d'identifier un ensemble de fichiers, par exemple pour créer une liste blanche accessible à un sujet. La création et la structure de données de cette liste sera détaillée dans la section 4.3.6.

```
1  int fs_walk_write(struct landlock_ctx_fs_walk *ctx)
2  {
3          if (ctx->cookie == 1 && ctx->inode_lookup &
               ↪ LANDLOCK_CTX_FS_WALK_INODE_LOOKUP_DOTDOT) {
4                  ctx->cookie = 0;
5          }
6          if (ctx->cookie == 0 && bpf_inode_map_lookup(&dirs, ctx->inode)) {
7                  ctx->cookie = 1;
8          }
9          return LANDLOCK_RET_ALLOW;
10 }
```

FIGURE 4.6 – Programme *Landlock* pour le parcours de dossiers

Le programme de la figure 4.6 teste si un des dossiers de l'arborescence d'un chemin, représenté par `ctx->inode`, est présent dans la liste `dirs`. Les métadonnées de ce programme, détaillées dans la section 4.3.3, définissent qu'il peut être chaîné sur des instances de lui-même en fonction des vérifications d'accès déclenchées par le noyau. La variable `ctx->cookie` permet de maintenir un état tout au long de la séquence d'interprétation de programmes correspondant au parcours de chacun des dossiers d'un chemin. La valeur initiale de la variable `ctx->cookie` est 0. Dans notre exemple la valeur 1 correspond à l'état indiquant que le fichier est compris dans l'arborescence, alors que la valeur 0 correspond à l'état contraire. Pour cet exemple, ce programme autorise les parcours d'arborescence de fichier mais nous verrons par la suite le programme qui complétera la règle pour restreindre l'accès au fichier identifié.

La figure 4.7 montre l'exemple de parcours du chemin `/data/write/../read/`. Lors de

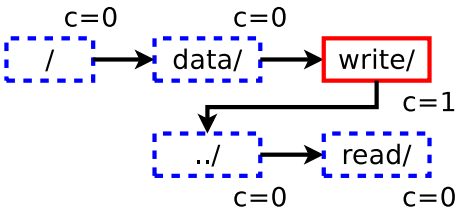


FIGURE 4.7 – Exemple de parcours d'un chemin avec un retour arrière

ce parcours, le programme `fs_walk` utilise la fonction `bpf_inode_map_lookup` pour vérifier si le dossier en cours d'évaluation est présent dans la liste `dirs`. Si c'est le cas, la règle passe dans l'état qui correspond à un fichier appartenant à une des arborescences de la liste (`ctx->cookie = 1`), comme c'est le cas pour le dossier `write`. Lors du parcours du dossier « `..` », représentant le dossier parent de celui dans lequel il est contenu, c'est-à-dire `data`, l'état de la règle est réinitialisé. Cette solution simple permet de valider l'accès à l'arborescence de `write` en refusant par défaut les retours arrière. Pour une raison de lisibilité, cet exemple ne considère pas tous les retours arrière légitimes, mais il est tout à fait possible de les prendre en compte en intégrant un compteur de profondeur d'arborescence avec `ctx->cookie`, comme le montre l'exemple fourni avec la huitième proposition de *Landlock*[2]. Un chemin est parcouru de manière similaire lors de la traversée de liens symboliques (*symlinks*). Dans cet exemple, la décision finale d'accès au fichier est déléguée au programme chaîné à `fs_walk` qui se base sur l'état transmis.

Lors de la création d'un programme de type *fs_pick*, le développeur peut faire en sorte qu'il soit chaîné avec la dernière interprétation d'un programme *fs_walk*, ce qui permet d'utiliser son état via la variable `ctx->cookie`. Il est ainsi possible de valider l'accès à un fichier en fonction de son emplacement dans l'arborescence du système de fichiers.

```
1 int fs_pick_write(struct landlock_ctx_fs_pick *ctx)
2 {
3         if (ctx->cookie == 1 && !(ctx->inode_lookup &
              ↪ LANDLOCK_CTX_FS_WALK_INODE_LOOKUP_DOTDOT)) {
4                 return LANDLOCK_RET_ALLOW;
5         }
6         return LANDLOCK_RET_DENY;
7 }
```

FIGURE 4.8 – Programme *Landlock* pour valider l'écriture dans un fichier

Les métadonnées du programme de la figure 4.8 stipulent qu'il est chaîné avec le programme `fs_walk` (figure 4.6) et qu'il va être interprété pour valider une écriture sur un fichier. L'état de la dernière interprétation du programme `fs_walk` est utilisé pour vérifier que le fichier demandé est un descendant d'un des dossiers de la liste `dirs`. Si c'est le cas, alors la requête est accordée ; dans le cas contraire, elle est refusée.

Analyse de l'approche

Pour qu'une telle règle soit efficace, il faut être capable de valider toute récupération d'une référence au système de fichiers, autrement dit à un descripteur de fichier. En effet, un appel système qui utilise un descripteur de fichier ne déclenche pas l'interprétation d'un programme de type *fs_walk* mais seulement de type *fs_pick*. Hormis l'ouverture d'un

2. https://lkml.kernel.org/r/20180227004121.3633-1-mic@digikod.net

fichier, deux autres moyens permettent de récupérer un descripteur de fichier : une réception via une IPC comme un socket UNIX, ou un héritage. Si c'est nécessaire, il est donc possible d'autoriser l'utilisation d'une liste blanche de fichiers avec la fonction `bpf_inode_map_lookup`. Comme nous le verrons dans la section 4.3.6, il est également possible de mettre à jour une telle liste à la volée.

Il est possible qu'un processus demande l'ouverture d'un fichier via un chemin relatif à son dossier courant ou relatif à un dossier préalablement ouvert, comme c'est le cas avec l'appel système `openat`. Ce cas est pris en compte par le type de programme *fs_ get*. Celui-ci est capable d'étiqueter des fichiers, avec la fonction `bpf_landlock_set_tag`, pour garder leur trace lors de leurs possibles utilisations futures. En effet, les programmes de type *fs_ walk* et *fs_ pick* sont capables de lire cette étiquette grâce à la fonction `bpf_inode_get_tag`.

Un autre cas particulier est la possibilité de modifier une arborescence via des points de montage. Ce type d'action correspond à une permission *Landlock* qui peut donc être validée par un programme.

L'avantage de vérifier les accès demandés lors de la récupération de références au système de fichiers est de limiter au strict minimum l'interprétation de programmes *Landlock*. L'impact sur les performances est ainsi minimal étant donné que les programmes ne sont pas interprétés lors des nombreux appels système liés à l'utilisation de fichiers, ce qui comprend les lectures et écritures, mais seulement lors de leur accès initial. D'un point de vue plus général, le chaînage de programmes *Landlock* minimaux offre une grande flexibilité tout en se calquant sur les principes de vérification d'accès actuellement effectués par Linux, comme pour le parcours de l'arborescence d'un fichier. Le nombre de paramètres d'entrée d'une règle, correspondant principalement à son contexte, offre également la possibilité d'implémenter un cache efficace pour limiter davantage l'interprétation de programmes.

4.3.6 Communication entre espace utilisateur et espace noyau

Pour définir une politique de sécurité, il faut être capable de décrire les objets dont l'accès doit être autorisé ou refusé. Cette définition des objets doit être faite par un processus utilisateur et utilisable par un programme *Landlock*. Une zone de partage de données doit donc être accessible entre l'espace utilisateur et l'espace noyau. Pour être capable de mettre à jour la politique de sécurité, il faut également pouvoir modifier ces données tout au long de l'utilisation d'un programme *Landlock*.

Map eBPF

Une *map* eBPF est un espace de stockage accessible à l'espace utilisateur via un descripteur de fichier, mais également accessible à partir d'un programme eBPF. Il existe différents types de *map*, suivant la manière de stocker les données (p. ex. tableau à une dimension, table de hash...). Ces espaces de stockage permettent de communiquer entre l'espace utilisateur et un programme eBPF en cours d'exécution, et également entre plusieurs programmes eBPF. Les actions sur une *map* à partir d'un programme eBPF sont possibles via l'appel à des fonctions (p. ex. lecture, mise à jour d'une entrée...). Le spectre des actions possibles est variable suivant le type de la *map* et celui du programme eBPF.

Un processus détenteur d'une *map* peut à tout moment s'en interdire l'accès en fermant le descripteur de fichier associé. La fermeture de tous ces descripteurs de fichier équivaut à interdire définitivement à l'espace utilisateur l'accès à une *map*.

Map inode Landlock

Un processus accède à des fichiers via des arborescences de fichiers sur lesquelles nous pouvons nous appuyer pour décrire une politique de sécurité. Une *map inode Landlock* contient

une liste de références à des fichiers ou des dossiers. Cette liste permet à un programme *Landlock* d'identifier la présence d'un fichier grâce à la fonction `bpf_inode_map_lookup`. Celle-ci retourne une valeur associée à la référence, qui peut être utilisée comme une étiquette.

La création d'une *map inode* s'effectue via l'appel système `bpf` et une commande dédiée. Cette *map* est initialement vide. Pour l'étendre avec un nouveau fichier, la première étape consiste à ouvrir un fichier ou un dossier pour obtenir un descripteur de fichier lui faisant référence. Ce descripteur est ensuite passé en argument à l'appel système `bpf` avec une commande dédiée. Le noyau stocke alors une copie de la représentation interne du descripteur de fichier, appelée *inode*. La combinaison de ce type de *map*, de la fonction `bpf_inode_map_lookup` et d'un programme *Landlock* de type *fs_ walk* permet d'exprimer des règles portant sur des arborescences complexes de fichiers.

4.4 Évaluation

Cette section détaille plusieurs points importants pour l'évaluation des objectifs de *Landlock*. Pour cela, nous nous appuyons notamment sur celle de *StemJail*, présentée dans la section 3.3.3.

4.4.1 Évaluation des performances

L'utilisation d'un moniteur de référence est nécessaire pour prendre des décisions de sécurité. Cependant, le fait d'avoir plusieurs moniteurs totalement indépendants, comme c'est le cas du noyau et potentiellement de plusieurs processus *StemJail*, implique des changements de contexte qui, bien que limités grâce au système de cache, peuvent s'avérer coûteux.

Contrairement à un moniteur en espace utilisateur comme utilisé dans *StemJail* [1, 2], l'approche de *Landlock* permet à un utilisateur de charger une règle de son choix qui sera prise en compte par un moniteur en espace noyau. Ceci permet d'éviter des dégradations de performance liées aux changements de contexte des moniteurs en espace utilisateur.

Étant donné que nous prenons en compte des utilisations non privilégiées, *Landlock* a été développé pour contrôler et limiter la consommation de ressources. La mémoire utilisée par les programmes *Landlock* est précisément comptabilisée et imputée à l'utilisateur qui les crée. De plus, la création et l'utilisation de ces programmes sont conçues pour limiter l'utilisation de la mémoire, ce qui permet de passer à l'échelle d'un grand nombre de programmes sur de nombreux processus.

Pour évaluer les performances de *Landlock*, notamment vis-à-vis de celles de *StemJail*, nous avons suivi la même méthodologie détaillée dans la section 3.3.3. La règle *Landlock* appliquée pour ces tests autorise tous les accès requis par les différentes applications exécutées mais différencie les accès en lecture des accès en écriture. Les tests de performance de la table 4.1 ont été réalisés sans utilisation de mécanisme de cache qui pourrait accélérer significativement *Landlock*, comme c'est le cas pour *StemJail*.

Le test Gunzip décompresse le fichier d'archive (`tar.gz`) correspondant à la version 4.4 de Linux. Ce test stresse le système de fichiers par des opérations de lecture de l'archive et des opérations d'écriture vers le fichier extrait. Excepté pour les premiers accès aux fichiers, notamment l'archive, il n'y a par la suite plus d'interprétation de règle. L'impact en termes de performance est négligeable, mais visible étant donné que les *hooks* de *Landlock* sont appelés pour toutes les demandes d'accès, que l'interprétation d'une règle soit nécessaire ou non, comme cela peut être le cas pour une lecture ou une écriture. La différence entre ces résultats pour un système de fichiers sur un disque mécanique et un autre en mémoire vive peut être due à des optimisations du cache du *ramfs*.

	Natif		*Landlock*			
Test	**HDD**	**RAM**	**HDD**		**RAM**	
Gunzip	11,48	6,36s	11,81s	2,9%	6,43s	1,1%
Untar	15,11s	1,57s	16,53s	9,4%	2,02s	28,7%
Zip	33,69s	32,13s	35,95s	6,7%	33,81s	5,2%
Build-1	1113,75s	1109,71s	1132,84s	1,7%	1141,51s	2,8%
Build-4	381,17s	364,44s	382,47s	0,3%	382,49s	5,0%

TABLE 4.1 – Résultats de tests de performance (Landlock patch v8)

Le test Untar consiste à extraire l'archive (`tar`) décompressée. Ce test montre un impact plus important sur les performances étant donné le grand nombre d'accès demandés. En effet, extraire de multiples fichiers sur le système de fichiers implique une interprétation des règles très fréquente. Cependant, la charge apportée par *Landlock* est plus faible vis-à-vis des opérations d'accès aux données dans le cas d'un disque mécanique, et est au contraire prépondérante dans le cas de l'utilisation d'un système de fichiers entièrement en mémoire vive.

Le test Zip crée une nouvelle archive (`zip`) à partir des fichiers provenant de l'archive originale. Ce test implique un grand nombre d'accès au système de fichiers comme pour Untar, mais nécessite le parcours et la lecture de tous les fichiers sources. Les calculs importants requis par l'opération de compression, absents avec Untar, forment un goulot d'étranglement plus important vis-à-vis des opérations de lecture du système de fichiers, même pour un *ramfs*.

Le test Build-1 consiste à compiler la version 4.4 de Linux sur un seul cœur logique du processeur (1 *job*). Ce test crée un grand nombre de processus à faible durée de vie qui accèdent à un grand nombre de fichiers différents. Les résultats montrent un impact négligeable lors d'une compilation avec l'application d'une règle *Landlock*. Le test Build-4 est également une compilation à partir des mêmes fichiers, mais en utilisant les 4 cœurs logiques du processeur (4 *jobs*). Ceci a pour conséquence de créer un grand nombre d'accès concurrents au système de fichiers, ce qui à un impact similaire à une compilation séquentielle. Ce comportement n'est pas problématique grâce à l'utilisation par le code noyau de *Landlock* du mécanisme de *Read-Copy-Update* (RCU), qui permet d'éviter le recours à des verrous qui pourraient bloquer l'utilisation d'une même *map* partagée entre plusieurs instances de règle appliquées à leur processus respectif.

Les performances de *Landlock*, sans mécanisme de cache, sont acceptables étant donné qu'il s'agit d'une couche supplémentaire de contrôle d'accès qui vient compléter les espaces de noms Linux.

4.4.2 Analyse de sécurité

Comme beaucoup d'autres interfaces du noyau, les fonctionnalités fournies par *Landlock* sont directement exposées à des processus malveillants. Cette section détaille les différentes menaces et les stratégies employées pour protéger le système.

Surface d'attaque et protection de la TCB

Landlock apporte ici une nouvelle interface au noyau Linux, pensée dès le départ pour du contrôle d'accès non privilégié sécurisé. Contrairement à d'autres mécanismes comme les espaces de noms utilisateur, les modifications apportées par *Landlock* sont réduites et donc

plus facilement auditables. De plus, il n'y a pas de changements de la sémantique concernant l'administration du système étant donné que *Landlock* ne fournit que des fonctionnalités de restriction d'accès qui s'ajoutent à celles déjà mises en place par l'administrateur.

Il est difficile de comparer des codes correspondant à une même fonctionnalité du noyau, comme c'est le cas pour les différents contrôles d'accès implémentés avec des modules de sécurité Linux. Concernant le LSM *Landlock*, le nombre de lignes de code (*Source Lines Of Code* : SLOC) ajoutées est de l'ordre d'un millier [3], ce qui est relativement peu et contribue à limiter la surface d'attaque du noyau.

L'architecture en place doit protéger contre toute modification du comportement de la TCB qui pourrait porter atteinte à la confidentialité ou l'intégrité des données auxquelles le processus n'a pas légitimement accès. En particulier, les données provenant de l'espace utilisateur ne doivent pas avoir d'impact en termes de sécurité sur la TCB du système. Tout accès aux données relatives à la mémoire du noyau doit être maîtrisé et soumis à validation. La liste des objets du noyau accessibles aux programmes *Landlock* doit donc être clairement identifiée par des interfaces facilement auditables. Comme vu dans la section 4.3.1, le LSM *Landlock* vérifie que les programmes *Landlock* utilisables respectent ces contraintes grâce au typage fort et aux contraintes sur l'utilisation de pointeurs. De même, le vérificateur eBPF permet de se protéger contre des attaques en déni de service qui peuvent provenir d'une utilisation exagérée de la mémoire ou d'un temps d'exécution non maîtrisé.

Utilisation optionnelle du JIT pour eBPF

Pour des raisons de performance, il est possible d'activer la compilation à la volée (*Just In Time* : JIT) des programmes eBPF. Le JIT implique d'allouer une zone mémoire dans un premier temps en écriture et dans un second temps en exécution. Dans le contexte d'eBPF, ceci permet donc à un processus en espace utilisateur de choisir indirectement un sous-ensemble d'instructions exécutables qui seront mises en place dans l'espace mémoire du noyau. Même si ce sous-ensemble d'instructions est contrôlé, le JIT peut grandement faciliter l'exploitation d'un bogue noyau. Si, suite à une première vulnérabilité, l'attaquant peut détourner le flot d'exécution du noyau, via des techniques comme le *Return-oriented Programming* (ROP), cela ne veut pas pour autant dire qu'il peut y insérer un code arbitraire. Afin qu'il puisse avoir une maîtrise suffisante pour faire une escalade de privilèges, par exemple en changeant les privilèges d'un de ses processus, il faut qu'il trouve une zone mémoire contenant des instructions exécutables par le noyau. Elena Reshetova *et al.* [47] mettent en avant cette problématique ainsi que les contre-mesures mises en place dans Linux. Parmi celles-ci, nous pouvons citer l'utilisation de zones mémoires en lecture-exécution pour le code généré, sa disposition pseudo-aléatoire et le *constant blinding*.

L'utilisation du JIT d'eBPF est optionnelle et désactivée par défaut. En termes de performance, l'interprétation du bytecode est d'environ 55% plus lente que l'exécution du code natif généré via du JIT [48]. L'utilisation de l'interpréteur peut donc être jugée satisfaisante.

Protection contre escalade de privilèges via processus supervisé

Limiter le contrôle d'accès à un rôle administrateur restreint la définition de la politique de sécurité à un seul point de vue et à un nombre de cas d'usages limité. Donner la possibilité à tous les utilisateurs d'apporter leurs propres restrictions d'accès leur permet d'étendre et de raffiner la politique de sécurité pour des cas d'usage *ad hoc*. Cependant, si nous considérons que les processus de ces utilisateurs non privilégiés peuvent inclure des processus potentiellement malveillants, les processus supervisés peuvent devenir des cibles.

3. contenu du dossier `security/landlock/`

Les applications usuelles ne sont pas conçues pour être exécutées dans un environnement malveillant. Tous les cas d'erreur ne sont pas pris en compte et un attaquant modifiant des contrôles d'accès peut induire un processus en erreur, ce qui peut mener à des attaques de type *confused deputy*. Il faut donc prendre en considération la protection des processus plus privilégiés qui peuvent recevoir des restrictions d'accès supplémentaires.

Afin de pouvoir appliquer de nouvelles restrictions sans remettre en cause les accès existants, une protection simple et efficace consiste à interdire toute restriction d'accès sur des processus qui ont plus de privilèges que celui qui souhaite les contraindre. Ces privilèges peuvent correspondre aux capacités du processus, mais également à son identité (utilisateur et groupes propriétaires), ou encore à son emplacement dans la hiérarchie des processus [4]. Le chargement de programmes *Landlock* par un processus non privilégié requiert que la propriété `no_new_privs` soit présente sur le ou les processus qui doivent se faire assigner un tel programme. Cette propriété apporte la garantie qu'un tel processus ne pourra pas gagner de privilèges supplémentaires. Par exemple, si un tel processus exécute un binaire SUID, l'exécution fonctionnera, mais ne changera pas l'UID du processus. Cette protection garantit qu'un processus capable de mettre en place des restrictions sur d'autres processus ne pourra pas faire d'escalade de privilèges via ceux-ci.

La protection mise en place par *Landlock* ne protège donc pas explicitement les processus supervisés, mais garantit que l'ajout de restrictions sur ces processus ne puisse pas mener à une escalade de privilèges.

Protection des sujets

Linux offre différentes fonctionnalités de débogage qui permettent de prendre le contrôle d'un processus ou de modifier sa mémoire, par exemple via l'appel système `ptrace` ou le système de fichiers *proc*. Ces fonctionnalités permettent donc de faire exécuter du code arbitraire à un autre processus et d'accéder à ses données, ce qui peut permettre d'outrepasser une politique de sécurité.

Afin de protéger les processus entre eux, *Landlock* apporte de nouvelles restrictions sur les actions qui peuvent servir à modifier le comportement d'un processus. *Landlock* complète les restrictions imposées de base par la politique de sécurité appliquée par Linux en s'assurant qu'un sujet *Landlock* ne puisse pas effectuer une escalade de privilèges en prenant le contrôle d'un autre sujet. Tous les programmes *Landlock* qui sont appliqués à un processus débogueur doivent également être appliqués à ses processus débogués. Par exemple, un processus en dehors d'une *sandbox Landlock* peut être autorisé à déboguer un processus placé dans une telle *sandbox*. De même, un processus parent d'un processus placé dans une *sandbox* peut également être autorisé à le déboguer.

Sur l'exemple de la figure 4.2b (page 54), les processus `P1` et `P2` ne sont pas cloisonnés ; ils sont donc autorisés à se déboguer l'un l'autre. Par contre, lorsque le processus `P1` s'applique un cloisonnement (sous-figure 4.2c), il n'est plus autorisé à déboguer `P2` qui a potentiellement moins de limitations. Le processus `P1` est par contre autorisé à déboguer ses autres fils : les processus `P3` et `P4`. Lors de la création du processus `P3` (sous-figure 4.2d), celui-ci est soumis aux mêmes limitations que son parent, mais il est autorisé à le déboguer. Cette autorisation n'est plus valable lorsque `P3` se cloisonne (sous-figure 4.2f).

Protection contre le contournement des autres contrôles d'accès

Dans un contexte non privilégié, il est important de prendre en compte les autorisations du processus qui met en place les règles d'accès aux fichiers. En effet, une nouvelle politique

4. C'est notamment le cas pour la restriction de débogage imposée par la fonctionnalité `ptrace_scope` du LSM Yama.

de sécurité *Landlock* s'ajoute aux politiques de sécurité déjà appliquées sur le système et il ne faudrait pas que l'utilisation d'une référence à un fichier permette d'inférer des informations jusque-là inaccessibles. Le mécanisme de *map inode* a été conçu pour éviter de permettre à un programme *Landlock* d'inférer des informations auxquelles le processus qui l'a chargé n'aurait pas accès. Par exemple, si un processus à l'origine d'un programme *Landlock* n'a pas le droit d'accéder aux métadonnées d'un fichier, alors ce programme doit avoir des restrictions identiques, même lorsqu'il est interprété pour valider l'accès aux données d'un tel fichier. Dans le cas contraire, il serait possible de créer un programme qui utiliserait une *map inode* pour outrepasser le contrôle d'accès sur les métadonnées des fichiers. Pour se protéger contre ce type d'attaque, d'une part l'ajout d'un fichier via un descripteur de fichier nécessite d'y avoir accès lors de son ouverture, et d'autre part les fonctions accessibles aux programmes *Landlock* ne permettent que de vérifier si un fichier accédé correspond à un fichier légitimement accessible. Seuls des fichiers dont l'accès est déjà autorisé peuvent donc être identifiés, ce qui rend implicite et assure le respect des contrôles d'accès du système.

4.4.3 Validation d'utilisation

Définition d'objets

Les espaces de noms Linux permettent la création d'environnements isolés, ce qui représente une forme de contrôle d'accès. Dans l'implémentation actuelle de *StemJail*, ce contrôle d'accès limite l'expressivité et l'application des règles.

La définition d'objets via les espaces de noms consiste à influer sur la visibilité de ces objets, ce qui représente un seul PDP et une seule action : l'ouverture d'objets. Pour un espace de noms de fichiers, la définition d'objets consiste à rendre visibles des arborescences de fichiers. Ces arborescences peuvent être rendues accessibles en lecture seule ou lecture-écriture. Le contrôle d'accès est uniquement possible au niveau de l'ouverture d'un fichier et non lors de son utilisation. Il n'est donc pas possible d'appliquer un contrôle une fois l'objet ouvert, comme c'est le cas par exemple pour l'utilisation d'un descripteur de fichier récupéré par un sujet qui ne l'a pas ouvert.

Une autre contrainte de l'espace de noms de fichiers est la granularité de définition des objets. Dans un espace de noms de fichier, l'utilisation des points de montage *bind* permet de définir un objet qui représente une arborescence de fichiers et non un fichier. Il n'est donc pas possible d'affiner l'accès à une sous-partie de cette arborescence uniquement via cet espace de noms. Par exemple, lorsqu'un processus souhaite accéder à un dossier légitimement autorisé, pour des raisons de compatibilité applicative, il est nécessaire qu'il puisse voir et parcourir l'arborescence de ses dossiers parents. Cependant, ceci n'est pas possible uniquement via un espace de noms de fichiers. Pour remédier à cette limitation, le moniteur de *StemJail* peut émuler la lecture de certains dossiers pour donner l'illusion que le dossier est partiellement accessible, alors qu'il ne l'est pas via l'espace de noms. Nous sommes ainsi capables de simuler une granularité plus fine que ce que permet nativement l'espace de noms de fichiers. Cette simulation apporte de la complexité à l'architecture de *StemJail* et impacte les performances lors de la lecture d'un dossier dont le contenu doit être partiellement accessible.

Contrairement aux espaces de noms qui ont pour but de permettre la création d'environnements indépendants, le but premier de *Landlock* est de faire du contrôle d'accès. Les problèmes rencontrés lors de la création de *StemJail* sont nativement résolus par *Landlock*. Par exemple, lors du parcours d'une arborescence de fichiers, un moniteur *Landlock* peut autoriser la vue du contenu d'un dossier tout en limitant l'accès à seulement une partie de ses fichiers. Ceci est possible par la définition plus fine des objets auxquels nous souhaitons

contrôler l'accès. Grâce à l'infrastructure LSM, nous pouvons définir des objets qui peuvent correspondre à la granularité des objets noyau. Nous sommes ainsi capables de contrôler l'utilisation des objets connus de *Landlock* à tout moment, ce qui permet par exemple d'interdire l'utilisation d'un descripteur de fichier.

Compatibilité applicative

StemJail tire parti de la fonctionnalité `LD_PRELOAD` pour s'intégrer de manière non intrusive dans les processus sous la coupe d'un moniteur. Ceci fonctionne très bien, et grâce à des optimisations de cache, donne des performances tout à fait acceptables. Nous pouvons cependant noter que l'utilisation d'une telle fonctionnalité peut introduire des incompatibilités pour certaines applications, notamment celles compilées statiquement, c'est-à-dire en intégrant la bibliothèque *libc* dans leur code, ou en utilisant directement les appels système. Cette limitation peut gêner l'adoption d'un système comme *StemJail* pour des applications non modifiables.

Landlock s'intègre dans le noyau via l'interface LSM dédiée à la validation de l'accès aux ressources du noyau. Bien qu'il soit possible de modifier une application pour qu'elle se place seule dans une *sandbox*, il est également possible de mettre en place une *sandbox* avant de lancer une application. Il n'y a donc pas besoin de modifier le code exécuté en espace utilisateur. Que les applications utilisent des bibliothèques partagées, qu'elles soient compilées en statique ou qu'elles utilisent directement les appels système sans passer par la *libc*, les vérifications nécessaires se feront de toute façon au niveau du noyau et donc de manière transparente. Avec l'utilisation de *Landlock* à la place des espaces de noms, nous n'avons donc plus de problème de compatibilité applicative.

Audit et délégation de l'écriture des règles

Contrairement à une politique de sécurité unique pour tout le système, la décentralisation des définitions des règles *Landlock* ne donne pas forcément une vue d'ensemble du contrôle d'accès. Cette décentralisation découle cependant du choix de déléguer la maîtrise des applications qui embarquent ces politiques. Cette liberté offre l'avantage de pouvoir déléguer la création et le maintien d'une politique de sécurité spécifique à une application aux développeurs qui en ont la responsabilité. Leur développement peut ainsi être indépendant de la mise à jour d'une politique globale qui pourrait freiner la validation des évolutions d'une telle application. Cette approche est particulièrement pertinente pour les systèmes d'exploitation développés de manière décentralisée, comme c'est le cas de GNU/Linux.

Exemple de gestionnaire de *sandbox*

Afin de démontrer l'intérêt de *Landlock*, un exemple de gestionnaire de *sandbox* minimaliste a été développé. La configuration est définie par les variables d'environnement `LL_PATH_RO` et `LL_PATH_RW`. La première variable contient une liste de chemins dont l'accès est autorisé en lecture seule et la seconde contient les chemins dont l'accès est autorisé en lecture et écriture. Les arguments indiqués sur la ligne de commande correspondent à la commande de l'application qui doit être lancée dans la *sandbox*.

La trace 4.1 montre l'exécution d'un *shell* dans une *sandbox* instanciée sur mesure. L'environnement ainsi créé permet l'accès en lecture au dossier `/data/` mais l'écriture n'est autorisée que dans le sous-dossier `/data/write/`. La création d'un fichier `/data/read/-file.txt` est bien autorisée en environnement non cloisonné mais est refusée dans la *sandbox*, conformément à la politique de sécurité configurée.

```
$ touch /data/read/file.txt
$ echo $?
0
$ LL_PATH_RO="/data:/usr:/lib:/lib64" \
>        LL_PATH_RW="/data/write:/dev/urandom:/dev/random:/dev/null" \
>        ./landlock1 /bin/sh
Launching a new sandboxed process
$ touch /data/read/file.txt
touch: cannot touch '/data/read/file.txt': Permission denied
```

TRACE 4.1 – Exécution d'un *shell* dans une *sandbox* minimaliste

4.5 Conclusion

Ce chapitre présente *Landlock*, un nouveau mécanisme de contrôle d'accès complémentaire à ceux déjà présents sur Linux. Cet écosystème gagne un moyen de sécuriser les applications pour limiter les répercussions de leurs exploitations. *Landlock* fournit les fonctionnalités nécessaires à l'implémentation de politiques de contrôle d'accès, mais permet surtout aux développeurs de construire, simplement et en minimisant les risques d'erreur, des logiciels capables de créer et de se placer dans une *sandbox*.

Notre approche consiste à ajouter le minimum de code au noyau tout en offrant un maximum de possibilités sans que cela ne puisse avoir d'effets indésirables sur la sécurité du système. Nous avons détaillé comment tirer parti des fonctionnalités offertes par *Landlock* pour cloisonner l'utilisation de logiciels en limitant une compromission potentielle à un sous-ensemble minimal de données. La partie dynamique du contrôle d'accès permet de le faire évoluer au fil du temps en fonction des données ou privilèges strictement nécessaires à une instance d'application, ce qui permet de mettre en œuvre une politique de sécurité comme celle de *StemJail*. Nous avons parcouru les différentes interactions entre l'espace utilisateur et le noyau avec l'implémentation de *Landlock* en tant que LSM. Enfin, une évaluation des performances et une analyse de sécurité ont permis de valider la viabilité de la solution.

L'implémentation actuelle permet d'effectuer un contrôle d'accès sur le système de fichiers, ce qui représente le sous-système le plus complexe pour le contrôle d'accès sous Linux. D'autres sous-systèmes comme le réseau et les IPC sont envisagés pour élargir les capacités d'un contrôle d'accès *Landlock*.

Landlock est en cours d'intégration dans la version de référence du noyau Linux afin qu'il soit disponible dans les prochaines versions. Un travail important avec la communauté a permis d'améliorer les concepts initialement proposés suite à de multiples revues de code. Une première présentation à Kernel Recipes [4] et des discussions avec des développeurs Linux, par listes de diffusions et lors d'une session dédiée à *Landlock* lors de la Linux Plumbers Conference [49], puis une présentation à Linux Security Summit [6] ainsi qu'au FOSDEM [7], ont permis de faire avancer le travail d'intégration *upstream*.

Deuxième partie

Interaction de confiance entre l'utilisateur et le système

Chapitre 5

État de l'art de la sécurité des interfaces homme-machine

Sommaire

Les ordinateurs avec interface graphique se sont démocratisés dans les années 1980. La gestion de données de classifications différentes avec ces machines a marqué le début des recherches sur des gestionnaires de fenêtres multi-niveau de confiance [50, 51]. Qu'il s'agisse de contraintes militaires, de l'administration de multiples systèmes dans une entreprise, de la signature de documents numériques ou encore de la gestion de données personnelles, la problématique du cloisonnement sécurisé des informations doit prendre en compte l'interface homme-machine (IHM). Aujourd'hui, l'impact de l'interface utilisateur est un sujet peu étudié, mais néanmoins crucial pour un système sécurisé multiusage.

Dans ce chapitre, nous dressons l'état de l'art de la sécurité des IHM. La section 5.1 passe en revue les principaux éléments matériels et logiciels qui peuvent faire partie d'une IHM d'un ordinateur de bureau, d'un ordinateur portable ou encore d'un smartphone. Le matériel comprend notamment les périphériques responsables de la saisie d'informations provenant de l'utilisateur, ainsi que ceux qui permettent l'affichage. Ensuite, la section 5.2 décrit des catégories d'attaques tirant parti de l'IHM, afin de fournir les bases nécessaires à la définition de critères de sécurité. Les techniques potentielles de protection contre ces attaques sont décrites dans la section 5.3. Nous étudions ensuite dans la section 5.4 le modèle de sécurité de différents systèmes qui prennent en compte l'IHM. Notre objectif n'est pas de comparer ces systèmes entre eux, mais d'en extraire des caractéristiques qui puissent servir de base à la formalisation de propriétés de sécurité d'une IHM.

5.1 Vue d'ensemble des composants d'une IHM

L'interface homme-machine est l'ensemble des éléments physiques et logiques qui permet à une personne de communiquer avec une machine. Ceci comprend donc un ensemble d'éléments matériels permettant de récupérer de l'information d'un système d'information, par exemple avec un écran ou une enceinte audio, ainsi que des capteurs appelés périphérique d'entrée utilisateur ou *Human Interface Device* (HID). Ils comprennent différents éléments physiques dédiés à l'utilisateur : clavier, souris, pavé tactile, zone tactile d'un écran, joystick, télécommande... Ces périphériques permettent d'acquérir des actions physiques de l'utilisateur qui sont ensuite transmises aux différentes couches logicielles du système. À cela s'ajoutent des spécifications d'utilisation pour permettre l'établissement d'un protocole de communication entre le monde physique et l'espace numérique.

L'interface graphique utilisateur est chargée d'exposer les données de manière intelligible à l'utilisateur. Chaque application peut être responsable de sa propre interface graphique et est capable de dessiner ce qu'elle souhaite sur les zones d'affichage qui lui sont dédiées. L'interface graphique peut également mettre à disposition des fonctionnalités supplémentaires pour une intégration plus homogène, par exemple des notifications qui peuvent être dessinées par une application dédiée à ce service.

La figure 5.1 met en évidence les interactions entre les différentes couches des mondes matériel et logiciel. Chacune des communications de cet exemple est effectuée via un protocole particulier. Les pilotes de périphériques peuvent communiquer avec les HID et l'écran via le processeur et la mémoire physique. Les applications et le compositeur graphique communiquent entre eux via un canal et un protocole dédié qui utilisent également la mémoire de la machine.

Le logiciel, qui s'exécute sur le processeur de la machine, peut communiquer avec les périphériques matériels par différents bus comme l'USB ou le PCI Express. Ensuite, la lecture de ces données par le pilote matériel, qui fait traditionnellement partie du noyau du système d'exploitation, se fait via des accès à la mémoire physique.

Un serveur d'affichage, ou compositeur graphique, est chargé d'agréger les différents dessins des applications graphiques pour ensuite les afficher sur la sortie graphique, ty-

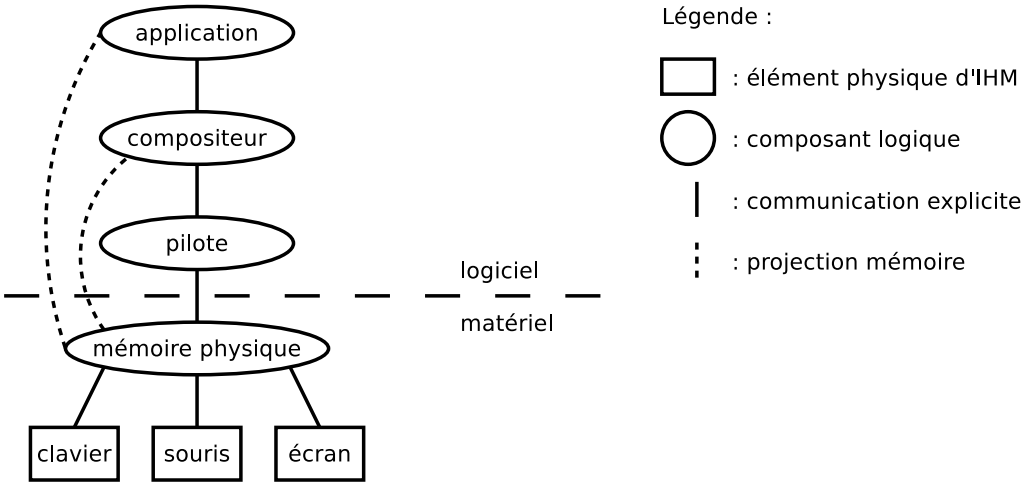


FIGURE 5.1 – Liens entre le matériel et les applications utilisateur

piquement un écran. Ce compositeur est également chargé de transférer les informations des périphériques d'entrée utilisateur, comme les frappes clavier et les mouvements de souris, aux applications. Ces échanges entre le compositeur et les applications s'effectuent grâce à un protocole dédié comme X Window System [52] ou Wayland [53]. Cette rupture protocolaire permet également d'enrichir ces données en y ajoutant des métadonnées comme la disposition des touches du clavier.

Dans le cas de l'utilisation d'applications de prise en main à distance via des protocoles comme VNC, une application peut également avoir le rôle de compositeur vis-à-vis de ces applications distantes. Le cas des navigateurs web est similaire, car ils interprètent et dessinent une page web distante, qui peut également être considérée comme une application graphique.

5.2 Critères de sécurité de l'IHM

Une interface homme-machine peut être sujette à différents types d'attaques. Chacune d'elles abuse d'une ou de plusieurs fonctionnalités requises par une interface utilisateur générique. Cette section passe en revue les attaques sur l'IHM, ce qui permet de définir les critères de sécurité de l'IHM. Des contre-mesures seront abordées dans la section 5.3.

Différentes recherches ont identifié des critères de sécurité à respecter pour différentes utilisations sécurisées d'une IHM. Yee [54] apporte notamment des définitions permettant d'identifier les rôles de composants d'IHM. Feske et Helmuth [37] décrivent des bonnes propriétés que doit vérifier un système sécurisé en y incluant son IHM. Enfin, Beckert et Beuster [55] formalisent des propriétés de sécurité pour un système simple dédié au vote. Nous proposons de définir de manière plus générique les critères de sécurité qui nous semblent pertinents sur une IHM sécurisée.

Nous excluons de notre périmètre d'étude les attaques liées au matériel, directement ou indirectement, comme l'exploitation du GPU, ainsi que les attaques par déni de service. De même, nous ne nous intéressons pas directement à la restitution correcte du contenu des documents [56] mais à l'interaction sécurisée de l'utilisateur avec différents domaines de sécurité tels que définis dans la section 2.2.5.

5.2.1 Sécurité des entrées utilisateur

Originalité

Un serveur d'affichage peut avoir comme fonctionnalité la simulation des entrées utilisateur. L'utilisation de cette fonctionnalité peut être légitime, par exemple dans le cas de claviers virtuels. De même, la prise de contrôle à distance d'un terminal utilisateur, par exemple pour de l'assistance utilisateur ou du travail à distance, peut être un cas légitime de simulation d'utilisateur physique. Les sujets utilisant cette fonction ont donc la possibilité de se faire passer pour l'utilisateur physique auprès des autres sujets.

La notion d'originalité des entrées utilisateur signifie qu'une entrée provient bien d'un utilisateur réel. Le respect de cette propriété permet de protéger les sujets récepteurs contre des attaques de type *confused deputy* [35] ayant pour but de se faire passer pour l'utilisateur. Ce type d'attaque peut par exemple consister à injecter des frappes clavier dans une application d'administration afin d'effectuer une escalade de privilèges.

Accomplissement

Dans le cadre de l'IHM, nous définissons comme accomplissement des entrées utilisateur la propriété qu'elles sont bien reçues par le sujet souhaité et que leur transfert est intègre. Par exemple, les attaques de type *user interface redress* ou *clickjacking* [57] cherchent à duper l'utilisateur pour lui faire effectuer des actions sur un élément de l'IHM autre que celui qu'il identifie.

Confidentialité

Dans le cas des interfaces utilisateur, les secrets qui peuvent transiter via l'IHM en provenance de l'utilisateur physique sont par exemple des mots de passe, mais également des conversations ou documents confidentiels, voire les commandes destinées à des applications. Nous pouvons également considérer certains périphériques tels que les caméras ou les microphones qui peuvent servir des usages contraires aux souhaits de l'utilisateur. Les enregistreurs de frappes (*keylogger*) représentent une menace à la confidentialité des entrées utilisateur. Le fait d'être capable d'intercepter toutes les frappes clavier peut toutefois être légitime, par exemple dans le cas d'un verrouillage de session.

5.2.2 Sécurité des sorties utilisateur

Affichage

Nous définissons l'affichage des sorties utilisateur comme la propriété qu'elles sont bien transmises à l'utilisateur physique. Ceci implique donc que l'utilisateur soit présent et apte à recevoir les données que lui expose un sujet. Une attaque contre l'affichage d'une sortie d'un sujet peut consister à en modifier l'intégrité ou tout simplement à bloquer le transfert d'une telle sortie vers le périphérique d'affichage qui en est la destination légitime (déni de service).

Attribution

Dans le contexte de la relation entre l'utilisateur et le système, nous définissons l'attribution des sorties utilisateur comme l'identification par l'utilisateur de tous les sujets qui ont participé à la modification de ces sorties. Comme décrit par Fischer *et al.* [58], la vérification de l'origine d'une sortie est nécessaire pour connaître la source d'une donnée retranscrite sur une sortie utilisateur par le système. Cette propriété permet de protéger

l'utilisateur contre des attaques de type *confused deputy* par un sujet ayant pour but de se faire passer pour un autre sujet.

La technique d'attaque *picture-in-picture* [59, 60] consiste à afficher et mimer l'interface d'une application sur l'espace graphique accessible. Cette technique permet de mettre en œuvre une attaque de type hameçonnage (*phishing*) pour récupérer des informations sensibles de l'utilisateur en le trompant sur le sujet à l'origine de la demande. Par exemple, une page web peut essayer de se faire passer pour une page d'un autre site en utilisant une représentation graphique similaire. Un autre exemple d'attaque appelée *login spoofing* consiste à se faire passer pour le gestionnaire de connexion, en se présentant de la même manière à l'utilisateur, de sorte à récupérer son mot de passe lorsqu'il le saisit.

L'attribution doit prendre en compte le temps d'affichage des informations pour que l'utilisateur puisse correctement identifier la source d'une sortie. Dans le cas contraire, il est possible de duper l'utilisateur avec une *race condition* entre l'information qu'il reçoit et l'action qu'il effectue. Par exemple, si une fenêtre modale (*pop-up*) surgit alors que l'utilisateur était en train de cliquer pour effectuer une action, il pourrait malencontreusement valider une action non souhaitée.

Confidentialité

Il ne doit pas être possible pour un sujet de se servir de l'IHM pour outrepasser une politique de sécurité qui contraint l'accès à des données. Par exemple, un sujet capable d'effectuer des captures d'écran (*screenshots*) a potentiellement accès à toutes les données qui sont présentées à l'utilisateur, indépendamment du contrôle d'accès contraignant leur lecture initiale.

Il existe également des attaques par canaux auxiliaires (*side channels*) comme la *UI state inference attack* [61] qui permet d'inférer des états de l'interface graphique, sans nécessiter de privilèges particuliers. Chen *et al.* expliquent que ce type d'attaque utilise des effets de bord de la mémoire partagée pour inférer des informations qui peuvent notamment être utilisées pour perfectionner d'autres attaques. Comme pour la partie précédente, nous excluons de notre périmètre les attaques par canaux auxiliaires et canaux cachés (*covert channels*).

5.3 Techniques de défense pour l'IHM

5.3.1 Contrôle d'accès du système

Le contrôle d'accès appliqué par le système constitue un moyen de protection des différents composants de l'IHM. Il peut par exemple s'agir du contrôle d'accès sur le matériel à partir de l'espace utilisateur. Pour limiter le nombre de sujets capables d'exploiter le matériel à des fins malveillantes, la restriction d'accès aux interfaces exposées par les pilotes représente une première barrière de protection. Par exemple, dans le cas de Linux, le contrôle d'accès sur les fichiers inclut les nœuds de périphériques. Ces nœuds sont représentés sous forme de fichiers dont l'accès peut être contrôlé de la même manière que des fichiers de données.

De manière similaire, les interfaces de communication entre les différents sujets de l'IHM peuvent être soumises à un contrôle d'accès. Ceci peut par exemple consister en des sockets UNIX ou tout autre IPC permettant de communiquer entre un client graphique et un serveur d'affichage.

5.3.2 Chemin de confiance

Comme l'explique Yee [54], pour communiquer à l'utilisateur des informations de confiance sur son interaction avec le système, il est nécessaire d'établir un chemin de confiance (*trusted path*) avec l'utilisateur. Ce chemin de confiance est un canal de communication, établi par un composant de confiance du système vers l'utilisateur final, dont l'intégrité est garantie et dont l'origine est connue de l'utilisateur. Il est crucial que l'utilisateur puisse discerner de manière fiable que l'information qu'il reçoit provient bien du système auquel il fait confiance afin de se protéger contre des attaques de type *confused deputy*.

Le principe de l'affichage de confiance peut être retrouvé dans les écrans des terminaux de saisie de mot de passe, par exemple lors de l'utilisation d'un *PIN pad* pour carte à puce, qui peut comprendre un écran pour visualiser un montant et un clavier pour valider le paiement. De la même manière, une simple diode électroluminescente ou un écran dédié peuvent répondre au besoin. Le facteur limitant est la quantité d'information communicable à l'utilisateur par ce chemin de confiance ainsi que l'espace physique occupé.

Comme discuté dans le chapitre 2, dans le cas de multiples activités et de la gestion de données de différents niveaux de confidentialité, il est nécessaire pour l'utilisateur de différencier ces activités. Il doit ainsi être capable de faire un choix éclairé sur le partage et la saisie de données.

Dans le cas d'un ordinateur multi-utilisateur, cette capacité à identifier de manière fiable les applications en cours d'utilisation est cruciale. En effet, il faut prendre en compte le fait qu'un utilisateur puisse induire un autre utilisateur en erreur avec une attaque par *spoofing*. Un exemple simple de *login spoofing* peut consister à simuler un écran d'authentification (*greeter*) pour récupérer le mot de passe d'un autre utilisateur. Une telle attaque peut passer inaperçue par la simulation d'une erreur de saisie de mot de passe, suivie de la déconnexion effective de l'application de simulation, et enfin de l'apparition automatique de l'écran d'authentification légitime.

5.3.3 *Secure Attention Key*

Une technique permettant de s'assurer d'une action particulière consiste à utiliser une *Secure Attention Key* (SAK) ou *Secure Attention Sequence*. Une SAK est une combinaison de touches qui ne peuvent être capturées que par le système de confiance, comme c'est le cas de la combinaison de touches `ctrl+alt+suppr` pour Windows ou de la combinaison de touches `ctrl+alt+impr+k` pour Linux. Pour une IHM de confiance, cette méthode est utile lorsqu'elle déclenche le passage dans un mode d'affichage dédié correspondant à un chemin de confiance. L'affichage ainsi présenté peut alors permettre d'effectuer des actions sensibles où l'utilisateur peut avoir la certitude d'interagir avec une IHM de confiance, comme dans le cas de Windows, détaillé dans la section 5.4.1.

Cette technique est intéressante, mais ne convient pas à tous les besoins. En effet, son usage nécessite d'être combiné à un chemin de confiance. Nous pouvons noter que l'usage de cette fonction n'est pas à l'initiative de l'application ou du système, mais à celle de l'utilisateur, ce qui limite son usage à une action volontaire de la part de l'utilisateur.

5.3.4 *Petname*

Un *petname* [62] est une étiquette secrète et unique, choisie par l'utilisateur et facilement mémorisable, par exemple une phrase ou une image. Ce concept vient du fait qu'un nom d'animal de compagnie peut être porté par plusieurs animaux, mais pour une personne, le nom de son animal de compagnie se réfère à un unique animal. Chaque *petname* est connu seulement de l'utilisateur et d'un système de confiance qui le relie par une relation bijective

avec une clef secrète. Cette étiquette permet à un système de confiance de représenter un élément privé à l'utilisateur.

S'il existe un secret partagé entre l'utilisateur et le système il est possible de communiquer des informations de confiance même dans un environnement partiellement de confiance. Cette hypothèse est vérifiée sous deux conditions :

— la confidentialité du secret est assurée ;

— et l'information à transmettre est indissociable du secret.

Une implémentation de ce concept correspond à l'utilisation d'un *petname* graphique. L'information à transmettre à travers un environnement non de confiance, mais pouvant assurer la confidentialité de transfert d'informations, doit alors être entremêlée avec le secret. Nous pouvons donc envisager d'ajouter un filigrane contenant l'information à l'image. Dans un environnement graphique où l'utilisateur n'est pas capable de dissocier une application de confiance d'une malveillante, l'affichage de cette image représente une référence de confiance. Cette méthode offre la possibilité à l'utilisateur de vérifier l'authenticité de l'affichage concerné, qui peut consister en une application ou un domaine de sécurité.

La limitation de cette approche est le manque de flexibilité et la limitation du nombre de secrets identifiables par l'utilisateur. En effet, pour identifier plusieurs domaines, il leur faut chacun un *petname* dédié connu de l'utilisateur. Il est donc complexe de gérer plusieurs domaines simultanément.

5.3.5 Contraintes sur le rendu graphique

Le contrôle des capacités d'affichage d'une application peut permettre à l'utilisateur d'identifier sans ambiguïté des éléments graphiques comme des boutons ou des zones de texte, mais également de limiter la liberté d'affichage comme le plein écran, l'affichage d'images ou de vidéos. Cet environnement graphique contraint a pour but de réduire fortement les moyens à disposition d'une application malveillante pour duper l'utilisateur, tout en ayant des éléments d'IHM génériques et utilisables.

Les contraintes imposées peuvent correspondre à l'utilisation exclusive d'une liste blanche d'éléments graphiques réutilisables, appelés *widgets*, comme des boutons ou des menus, et d'emplacements visuels identifiés comme ne pouvant pas induire l'utilisateur en erreur. Il est ainsi possible de définir des politiques de sécurité portant sur l'IHM et plus particulièrement l'affichage.

5.3.6 *Powerbox* et UDAC

Une *powerbox* [63] est une interface permettant à l'utilisateur d'accorder pour un sujet cloisonné l'accès à des ressources qui lui sont autrement inaccessibles. Cette demande d'accès est cependant à l'initiative du sujet. Dans un environnement graphique, une *powerbox* se présente habituellement à l'utilisateur sous la forme d'une boîte de dialogue. Un cas d'usage typique est une demande d'accès à un fichier. Un processus restreint appelle la procédure d'ouverture de fichier qui va proposer à l'utilisateur de sélectionner un fichier parmi tous ceux qui lui sont accessibles. Le fichier ainsi choisi sera rendu accessible au sujet initiateur de la requête. De cette manière, les contraintes liées au cloisonnement ne sont ni perçues par l'utilisateur ni par un sujet légitime qui utilise cette interface. Par contre, un sujet malveillant qui essaierait d'accéder à une ressource en dehors de son espace cloisonné ne pourrait pas effectuer un accès direct et serait donc contraint à faire une demande explicite à l'utilisateur. Il est bien sûr toujours possible de faire de l'ingénierie sociale auprès de cet utilisateur pour qu'il autorise l'accès.

Le concept du *User-Driven Access Control* [64, 60] (UDAC) est une généralisation du principe des *powerbox*. UDAC fait appel à des *Access Control Gadgets* (ACG), qui

représentent des *widgets* d'action. Nous retrouvons également des idées similaires dans EWS, présenté dans la section 5.4.2. Un ACG peut par exemple représenter un bouton pour enregistrer le son ambiant ou prendre une photo au moment de l'approbation, pendant une période définie, ou encore de manière permanente. Cette approche est intéressante, car elle permet d'intégrer de manière transparente de la délégation d'accès par l'utilisateur pour ses applications. Être capable d'attribuer des actions sur une application non de confiance requiert néanmoins un chemin de confiance entre le serveur d'affichage et le composant graphique qui reçoit cet événement. Il faut également être prudent quant à l'interprétation que peut avoir l'utilisateur en fonction du contexte d'utilisation. La description du *widget* doit donc être explicite et autosuffisante.

5.4 Aperçu de différents systèmes d'IHM

Dans cette section nous parcourons une liste non exhaustive de logiciels et protocoles parmi les plus fréquemment utilisés ou les plus appropriés d'un point de vue de la sécurité. Pour chacun d'eux, nous détaillons leur modèle de sécurité et les protections associées. Ceci a pour but d'identifier des caractéristiques de sécurité pertinentes.

Contrairement à la section 1.2.3, nous n'abordons pas les IHM de certains systèmes populaires comme iOS ou Android. En effet, leurs interfaces n'offrent pas de nouvelles propriétés de sécurité significatives vis-à-vis des systèmes ci-dessous.

5.4.1 Serveurs d'affichage communément utilisés

X Window System

Le protocole *X Window System* est apparu en 1984. Étant l'un des premiers protocoles de rendu, il est rapidement adopté par différents systèmes UNIX. Ce protocole fournit des primitives permettant de dessiner sur l'écran, d'interagir avec ces éléments graphiques et de gérer les entrées utilisateur. Parmi les différentes implémentations de serveur, identifié comme compositeur sur la figure 5.1, *X.Org* est aujourd'hui la plus utilisée.

Le modèle de sécurité d'un serveur consiste principalement à imposer des restrictions sur les connexions des clients [52]. L'authentification du client peut avoir lieu de plusieurs manières. Au niveau du réseau, il est possible de faire reposer l'authentification sur un tiers de confiance en utilisant le protocole Kerberos. Une autre méthode permet de se passer d'intermédiaire en utilisant un secret partagé. Ce secret se présente sous la forme d'un fichier contenant un *cookie*, c'est-à-dire une valeur secrète fournie par le serveur. Dans un fonctionnement conventionnel, un utilisateur du système est autorisé à ouvrir une session graphique, ce qui a pour conséquence de lui accorder le monopole d'utilisation des périphériques de l'IHM. Ce monopole est assuré par la création d'un nouveau *cookie* par le *login manager* lors de l'ouverture de session. À la fermeture de session, le serveur d'affichage est réinitialisé, ce qui invalide le *cookie* précédemment utilisé et révoque donc les accès au serveur d'affichage.

Une fois connectée à une session, l'extension *SECURITY* permet de définir deux domaines : *SecurityClientTrusted* et *SecurityClientUntrusted*. Ces domaines séparent les applications de confiance des autres. Le domaine *SecurityClientTrusted* n'a aucune restriction et peut donc accéder à toutes les ressources du serveur d'affichage, ce qui revient par exemple à lire et écrire les entrées et sorties utilisateur ou encore à faire des captures d'écran. Les applications du domaine *SecurityClientUntrusted* ne sont pas autorisées à lire le contenu des fenêtres de l'autre domaine ou à lire sans restriction les entrées utilisateur. Il est important de noter que ces contraintes sont respectées par le protocole de base, mais pas forcément par ses extensions, ce qui peut rendre perméable le domaine *SecurityClientTrusted*. En

pratique, nous avons constaté qu'il existe peu d'applications capables de s'exécuter sans défauts d'affichage dans le domaine *untrusted*. Cette restriction vient notamment des *toolkits* graphiques utilisés, comme c'est le cas de GTK et Qt, qui utilisent des extensions comme *RANDR*, incompatibles avec les contraintes du domaine *SecurityClientUntrusted*.

Le *framework X Access Control Extension* (XACE) est une généralisation de l'extension *SECURITY*. Il permet par exemple d'utiliser une politique SELinux [65] pour étiqueter des fenêtres ou restreindre le copier-coller. Nous pouvons cependant noter que l'implémentation de ce *framework* n'est pas complète [52] étant donné le manque de points de contrôle et en raison des extensions qui ne sont pas prises en compte.

En raison de son historique, le code du serveur et de ses extensions est par ailleurs très complexe. Avec ses 26 extensions, son interface de programmation (API) fournit 1278 points d'entrée [66]. La surface d'attaque exposée par ce composant critique est donc très importante.

Wayland

Le protocole et la bibliothèque Wayland ont officiellement été publiés en 2012. C'est un protocole pour compositeur graphique, c'est-à-dire pour un serveur d'affichage qui agglomère les images générées par ses clients et assure également le rôle de gestionnaire de fenêtres. En effet, chaque client doit créer entièrement son rendu, avec l'aide éventuelle de bibliothèques graphiques, puis le projeter en mémoire, représenté *in fine* par une fenêtre. Ce rendu est ensuite composé avec celui des autres clients pour former l'image finale des écrans. Contrairement au protocole X, Wayland ne donne aucune primitive de dessin et ne reconnaît que certains formats d'image correspondant aux fenêtres des applications (*surfaces*). Wayland est relativement indépendant envers ses clients. En effet, chaque client est responsable de son rendu et ne peut pas interférer avec celui des autres. Nous pouvons noter que l'API de Wayland est approximativement 15 fois plus réduite que celle de *X.Org* [66, 53].

Un compositeur qui implémente Wayland optimise les échanges entre ses clients et le dispositif d'affichage, notamment grâce à l'utilisation des primitives du processeur graphique. Ce protocole permet également au compositeur d'identifier précisément le client qui est responsable de chaque partie de l'image finale. De plus, les entrées utilisateur sont uniquement relayées au client qui est focalisé.

Weston est le compositeur de référence du projet Wayland. Il met en œuvre ce protocole et l'étend pour fournir des interfaces de communication entre les applications et le bureau. Il comprend trois types de compositeurs : système, session et embarqué.

Le compositeur système doit se lancer au démarrage de la machine. C'est lui qui doit notamment permettre d'afficher la progression du démarrage des services système. Lorsque le démarrage est terminé, le compositeur système doit afficher le gestionnaire de connexions utilisateur. Il peut également être chargé d'afficher la console et ses terminaux virtuels.

Le compositeur de session est lancé avec l'identité de l'utilisateur qui s'authentifie auprès du gestionnaire de sessions. Ce serveur est chargé d'afficher le bureau, de gérer les fenêtres ainsi que certains services utilisateur comme le verrouillage de l'écran.

Les applications utilisateur peuvent également avoir un compositeur embarqué. Il peut par exemple servir à afficher des *applets* dans un navigateur indépendamment du *plugin* utilisé.

Microsoft Windows

L'interface graphique du système d'exploitation Windows est majoritairement en espace noyau, ce qui comprend le gestionnaire de fenêtres, les menus des applications ou encore

les polices de caractères. La TCB du système est donc très importante, ce qui induit une grande surface d'attaque.

La sécurité de l'interface graphique repose sur plusieurs primitives, dont les *window stations* qui comprennent des *desktops* regroupant un ensemble de clients graphiques [67]. Les *desktops* sont isolés entre eux au niveau de certains composants de l'IHM, ce qui inclut notamment les événements graphiques, les frappes clavier ou l'affichage. Par contre, ce composant n'est pas suffisant pour cloisonner les processus efficacement. Tous les *desktops* de la *window station* interactive, nommée WinSta0, peuvent chacun leur tour afficher à l'écran et recevoir les entrées utilisateur. La *window station* WinSta0 comprend par défaut trois *desktops* : le gestionnaire de sessions (Winlogon), le bureau utilisateur (Default) et l'écran de verrouillage sécurisé (ScreenSaver). Chacun d'eux permet d'accéder à des fonctionnalités particulières comme le *User Account Control* pour Winlogon.

L'utilisation des *jobs objects*, des *tokens* et des niveaux d'intégrité, particulièrement l'*User Interface Privilege Isolation* (UIPI), permettent de contrôler l'utilisation du presse-papiers et de limiter l'accès à certaines ressources comme les fichiers ou les connexions réseau. Toutes ces fonctionnalités sont accessibles pour les développeurs, mais sont rarement utilisées [68].

L'*User Account Control* [31] (UAC) permet de demander de manière interactive à l'utilisateur, après validation de l'administrateur, de fournir des permissions à une application qui en ferait la demande. L'activation de cette fonctionnalité change le mode de l'interface de Windows, ce qui a pour conséquence de refuser toute manipulation de l'interface graphique par une application non privilégiée.

L'*Authentic User Gesture* (AUG) [69] est un système de *powerbox* qui permet d'identifier les actions de l'utilisateur via l'utilisation de boîtes de dialogue maîtrisées par le système (*Graphical Pipes*). Lorsque l'utilisateur effectue une action, comme la sélection d'un fichier, le résultat est transmis à l'application initiatrice. Ceci est similaire aux travaux de Roesner *et al.* [64, 60] détaillés dans la section 5.3.6. Cette fonctionnalité est renforcée par l'*App Container* qui permet de cloisonner des applications.

5.4.2 IHM logicielles sécurisées

Compartmented Mode Workstation (CMW)

Le prototype du projet *Compartmented Mode Workstation* [50, 51] a été initié par la *Defense Intelligence Agency* des États-Unis pour manipuler de manière sécurisée des données de différents niveaux de sensibilité sur un même poste utilisateur, formalisé par le modèle Bell et LaPadula [10]. L'utilisateur est considéré de confiance et est autorisé à accéder aux données correspondant à un niveau inférieur ou égal à son niveau d'habilitation.

Une étiquette de sécurité est assignée à chaque objet et à chaque sujet qui nécessitent une protection. Cette étiquette a pour but de fournir à l'utilisateur, ou aux processus qui s'exécutent en son nom, une information permettant d'identifier clairement un niveau de sensibilité. Le gestionnaire de fenêtres est ainsi capable d'afficher le niveau de sensibilité de l'information qui est dessinée dans une fenêtre. Certaines applications privilégiées sont autorisées à afficher le niveau de sensibilité plus finement que le contenu complet d'une fenêtre. Par exemple, un éditeur de texte conçu sur mesure peut étiqueter chaque paragraphe d'un même texte avec des niveaux de sensibilité différents.

Par défaut, le gestionnaire de fenêtres et toutes les fenêtres ouvertes se voient assigner le niveau de sensibilité maximal autorisé pour l'utilisateur. Le niveau d'un sujet créé à partir de cette fenêtre hérite de son niveau de sensibilité, ce qui permet au sujet de créer des documents avec ce niveau et d'accéder à des données d'un niveau inférieur ou égal. Pour créer des données destinées à un niveau inférieur au niveau maximal qui lui est autorisé,

l'utilisateur doit créer une nouvelle fenêtre assignée à ce niveau. Les données affichées dans cette fenêtre ne peuvent alors que correspondre à des niveaux de sensibilité inférieurs, et la création de documents correspondant au niveau de la fenêtre lui est autorisée.

Afin de protéger le système contre le *login spoofing*, des paramètres d'affichage immuables sont configurés de telle sorte que l'écran d'authentification de l'utilisateur soit discernable grâce à une taille de police plus grande que celles utilisées dans les fenêtres des applications, mais également par le fond d'écran blanc contrairement au gris utilisé lorsque l'utilisateur est authentifié. En complément, un sujet non privilégié n'est pas autorisé à dessiner en dehors de la fenêtre qui lui est assignée.

TRW Trusted X Window System (TX)

Le gestionnaire de fenêtres TRW Trusted X Window System [70, 71] (TX) est une alternative partielle au système CMW. TX apporte une protection contre les attaques de type *spoofing* de fenêtre qui visent à tromper l'utilisateur avec le dessin d'une étiquette par une application malveillante. Le gestionnaire de fenêtres TX identifie les fenêtres en leur associant une étiquette visible sur leurs bordures.

Étant donné la complexité de sécuriser un serveur X, l'approche de TX consiste à instancier un serveur d'affichage par niveau de sensibilité. Leurs rendus sont ensuite agrégés par le gestionnaire de fenêtres. Afin de mieux refléter le passage d'un niveau à l'autre, un changement de couleur des niveaux non actifs est appliqué.

Une zone de l'écran est réservée à l'affichage d'un menu fourni par un *shell* de confiance. Ce menu, non accessible et invisible des autres applications, met à disposition de l'utilisateur différentes fonctionnalités sensibles telles que le verrouillage de sa session ou le changement de son mot de passe. Cette zone affiche également à l'utilisateur le niveau destinataire des entrées utilisateur.

Enfin, TX a été conçu avec un minimum de code pour limiter l'impact sur la TCB.

DOpE

DOpE (Desktop Operating Environment) [72, 73] est un gestionnaire de fenêtres, initialement conçu pour l'OS temps-réel DROPS, permettant de limiter l'impact des applications sur les ressources du système. Il a pour but d'exécuter la partie graphique des applications en temps-réel en leur garantissant de la bande passante vers les composants graphiques.

Ce serveur adopte une approche contraignante de communication avec ses clients [74] via l'utilisation de *widgets* graphiques. Ne pas pouvoir afficher d'images arbitraires limite les possibilités d'affichage des clients, ce qui peut également renforcer la confiance dans le rendu d'une application. En effet, une telle contrainte pourrait être utilisée pour n'autoriser l'affichage d'une image donnée qu'à certaines applications de confiance, permettant ainsi à l'utilisateur de les identifier de manière fiable.

EROS Window System (EWS)

La sécurité du système Extremely Reliable Operating System (EROS) est basée sur l'utilisation de capacités. Le gestionnaire de fenêtres EROS Window System [75] (EWS) s'efforce de suivre plusieurs principes de sécurité dont l'isolation des clients, la maîtrise fine des ressources du serveur d'affichage ou encore de celle des entrées utilisateur.

Afin de réduire la TCB, les clients d'EWS sont chargés de leur propre rendu. Cela permet également de faire reposer l'impact des calculs nécessaires aux dessins du rendu sur chaque client. Un autre avantage de cette architecture est que le serveur n'est pas responsable de la gestion de ressources graphiques comme les polices de caractères.

Il est possible d'avoir un découpage hiérarchique de clients en sessions pour limiter l'utilisation de l'affichage par session. Les sous-sessions doivent être gérées par des applications de confiance comme le gestionnaire de fenêtres.

EWS applique un contrôle d'accès sur les transferts d'information via le copier-coller, ce qui inclut également la notion de temps d'exposition minimum de la commande à l'utilisateur. L'autorisation de recevoir un tampon de copie n'est accordée que lorsque le menu permettant de coller apparaît suffisamment longtemps à un même emplacement. De plus, l'action est effectuée seulement lors du relâchement de la zone d'action par l'utilisateur. Ces propriétés permettent d'éviter des attaques de type *clickjacking*.

L'opération de glisser-déposer est gérée par le serveur. Lorsqu'il détecte une telle action, il en informe le client source pour recevoir les données. Il les transmet ensuite au client de destination lorsque l'utilisateur finit l'action de glisser-déposer.

Une des propriétés importantes d'EWS est l'identification de confiance des fenêtres. L'affichage du titre et des bordures d'une fenêtre est effectué par le serveur. Pour accentuer la visualisation de ces propriétés, les fenêtres qui n'ont pas la focalisation sont assombries de manière à avoir un contraste élevé avec les étiquettes de la fenêtre qui est focalisée.

La problématique de la saisie de données d'authentification est résolue par un composant sûr permettant à une application de savoir si l'utilisateur physique s'est authentifié auprès du système, sans que l'application ait connaissance des éléments secrets. Afin que ce composant de demande de mot de passe soit reconnu par l'utilisateur, EWS met en place une session cliente qui a pour effet d'ajouter un calque de couleur rouge sur toutes les autres fenêtres et de l'indiquer dans une zone réservée en bas de l'écran ce qui correspond à un chemin de confiance.

Clip OS

Clip OS [8] est une métadistribution sécurisée basée sur GNU/Linux et développée par l'ANSSI [76]. La version poste utilisateur de ce système repose sur une architecture multi-niveau (*Multilevel Security*). Il est ainsi possible d'utiliser deux environnements utilisateur de niveaux différents : le niveau haut pour les données sensibles et le niveau bas pour des données moins sensibles. Le niveau bas est connecté au réseau local et donc le plus souvent à Internet. Le niveau haut est confiné à un réseau privé virtuel (VPN) correspondant à la sensibilité du niveau.

Un niveau utilisateur est représenté sous forme d'un bureau et d'un gestionnaire de fenêtres dédié affiché sur la majorité de l'écran. Le cloisonnement entre niveaux est mis en place avec les espaces de noms Linux ainsi que le mécanisme de cage fournit par Linux-VServer étendu avec une gestion des privilèges propre à Clip OS. Chaque niveau est rendu visible à travers une connexion VNC locale dédiée et invisible pour l'utilisateur. Cette rupture protocolaire sert à isoler chaque niveau du socle système, mais est également utilisée en tant que défense en profondeur dans l'éventualité de la compromission d'un niveau utilisateur. En complément, chaque client VNC dédié à un niveau appartient à un domaine de sécurité dédié, instancié par *X.Org* et comparable au domaine *SecurityClientUntrusted* décrit dans la section 5.4.1.

L'utilisateur bascule d'un niveau à l'autre à l'aide d'une barre de confiance, située sur le bord gauche de l'écran, qui affiche également le niveau en cours d'utilisation. Cette barre est une implémentation de chemin de confiance, c'est-à-dire qu'elle n'est ni accessible ni recouvrable par les applications des niveaux utilisateur. Elle peut également permettre d'accéder à des fonctionnalités limitées d'administration ou de supervision du système. Ces fonctionnalités offertes par le socle du système ne sont pas accessibles aux niveaux utilisateur. Pour chacun des niveaux utilisateur ainsi que pour le socle, la barre de confiance change de couleur pour refléter le niveau qui reçoit les entrées utilisateur.

Le transfert d'informations est fondé sur le modèle de sécurité de Bell et LaPadula [10] : il est possible d'envoyer des fichiers du niveau bas vers le niveau haut via une diode dédiée. Il est également possible de chiffrer des données du niveau haut vers le niveau bas, de sorte à préserver la confidentialité du transfert. Chacune de ces étapes est initiée et soumise à validation de l'utilisateur par une interface dédiée à ces opérations. Que ce soit pour la validation de transfert ou pour la saisie du mot de passe nécessaire au chiffrement, la barre de confiance reflète l'identité de l'application de saisie ou de confirmation, ce qui correspond donc à une *powerbox*.

Genode & Nitpicker

Le *framework* de système d'exploitation Genode [77] est issu des travaux sur la famille de micronoyaux L4. Il rassemble les composants nécessaires pour créer un système qui prend en compte les liens entre un micronoyau et la gestion des services offerts par le système. De manière similaire à EROS, le principe de l'isolation des services repose sur la hiérarchie des serveurs. Chaque processus client peut dialoguer avec les services de son parent qui contrôle la validité des demandes. Si le service recherché n'est pas présent parmi ceux mis à disposition par le parent, alors la requête est transmise à son ancêtre direct jusqu'à trouver le service recherché. Chaque service peut ensuite accepter ou refuser le client.

La partie graphique comprend Nitpicker [37] pour le serveur d'affichage et DOpE comme gestionnaire de fenêtres. Nitpicker a pour rôle de donner une vue virtuelle de l'écran, sous forme de *frame buffer*, et des entrées utilisateur aux applications. Il permet d'affecter des étiquettes de sécurité aux zones de l'écran utilisées. Ces étiquettes permettent d'identifier le processus affichant les informations ainsi que ses parents. Cette hiérarchie repose sur le nom des exécutables avec éventuellement un indicateur permettant de différencier plusieurs instances de hiérarchies similaires. L'utilisateur est ainsi capable d'identifier les domaines de sécurité représentés par cette hiérarchie. Par ailleurs, il est possible d'avoir plusieurs serveurs d'affichage imbriqués les uns dans les autres afin de contraindre plus fortement encore les domaines graphiquement.

Nitpicker a également été conçu pour résister aux attaques par déni de service. Pour cela, il repose sur l'utilisation des ressources de ses clients. En effet, le système permet à un processus d'allouer de la mémoire pour un autre processus. Cette zone mémoire est référencée comme appartenant au processus qui l'alloue, mais peut être utilisée exclusivement par celui qui la reçoit. En complément, le temps d'exécution des fonctions de dessin est maîtrisé par DOpE [72, 73], ce qui permet de se protéger contre des attaques temporelles (*timing attacks*). Enfin, Nitpicker est très simple ce qui permet de réduire la TCB et d'effectuer un audit de sécurité réaliste.

Qubes OS

Qubes OS est une distribution GNU/Linux/Xen dédiée à l'unique utilisateur du système, également administrateur de la machine. L'hyperviseur n'est qu'une partie du mécanisme de cloisonnement de Qubes OS. Chaque machine virtuelle, associée à un domaine de sécurité, communique avec le moniteur, implémenté par l'hyperviseur et une machine virtuelle d'administration. Un domaine a son propre espace de stockage et peut avoir une configuration réseau dédiée, par exemple contrainte à l'utilisation d'un VPN.

Les applications de chaque domaine sont identifiées par une décoration de fenêtre spécifique, caractérisée par un titre de fenêtre et une couleur dédiée, permettant à l'utilisateur de les différencier de manière fiable. Les modifications apportées à X incluent un protocole de communication robuste pour composer l'affichage des domaines d'utilisation sans qu'ils ne puissent avoir accès ni aux sorties ni aux entrées utilisateur des autres domaines. L'interface

utilisateur permet également d'effectuer des transferts d'information comme le copier-coller entre différents domaines.

TrustGraph

TrustGraph [78] est un gestionnaire de fenêtres basé sur le compositeur DirectFB. TrustGraph se base sur les domaines des applications tels que définis par sHype [79]. L'affichage de chaque domaine est étiqueté et est utilisé pour appliquer des règles sur la superposition de fenêtres, mais également pour appliquer un contrôle d'accès sur la capture d'écran. Celle-ci ne capture que les fenêtres du domaine courant. De même, la capture des entrées utilisateur n'est autorisée qu'au domaine actif. Le copier-coller est également restreint par une politique de sécurité.

5.4.3 IHM matérielles sécurisées

Interactive Link (Starlight)

Le projet Starlight [80] fournit une base matérielle pour faire du multi-niveau. La partie Interactive Link est un périphérique connecté à un poste de travail, qui a pour but de donner à l'utilisateur un espace de travail contenant des fenêtres d'applications d'un niveau haut et d'autres d'un niveau bas. Chaque niveau est isolé de l'autre, mais ce qui est affiché à l'utilisateur est une composition des fenêtres des deux niveaux. Lorsque l'utilisateur veut interagir avec un niveau, il commute un interrupteur physique pour changer le niveau récepteur des entrées utilisateur comprenant un clavier et une souris. Il est possible de connaître le niveau récepteur des entrées utilisateur avec un affichage de confiance. Celui-ci consiste en une suite de barres lumineuses s'allumant uniquement lorsque le niveau qui leur est associé est sélectionné. Nous pouvons noter que la problématique de HID malveillant ou vulnérable n'est pas prise en compte.

Cross Domain Desktop Compositor

Le système *Cross Domain Desktop Compositor* [81] est une implémentation principalement matérielle permettant de composer l'affichage de plusieurs ordinateurs tout en mutualisant un clavier et une souris. Sur chaque ordinateur, correspondant à un domaine de sécurité, un logiciel transmet dans le flux vidéo de l'écran des informations permettant d'identifier les fenêtres et menus des applications. Ensuite, un équipement similaire à un switch KVM (*Keyboard, Video and Mouse*), agrège les différents flux vidéo des ordinateurs connectés et les affiche sur un écran. Chaque fenêtre et menu est affiché entouré par une bordure de couleur dédiée à son domaine. Un seul domaine est capable de recevoir les entrées utilisateur à un instant donné. Ce domaine focalisé, identifié dans une zone dédiée de l'écran, peut être changé vers un autre domaine lorsque l'utilisateur clique sur une fenêtre de ce domaine.

5.5 Conclusion

Dans ce chapitre, nous avons identifié et décrit les différents éléments logiciels et matériels qui forment une interface homme-machine. Nous avons ensuite identifié trois critères de sécurité concernant les entrées utilisateur : l'originalité, l'accomplissement et la confidentialité. Nous avons également identifié trois critères de sécurité pour les sorties utilisateur : l'affichage, l'attribution et la confidentialité. Afin de contrer certaines attaques, nous avons listé des techniques de défense ainsi que leur utilisation par plusieurs implémentations de serveurs d'affichage. Nous retrouvons des similitudes entre tous les systèmes étudiés,

mais également des prises en compte de menaces différentes. Le chapitre suivant a pour but d'apporter un modèle permettant d'exprimer des propriétés de sécurité sur l'IHM et de définir une exécution sécurisée. Nous proposons ensuite des mécanismes permettant d'implémenter de telles mesures de protection.

Chapitre 6

Sécurité de l'interface utilisateur

Sommaire

Dans les différentes interfaces homme-machine (IHM), la restitution visuelle des données est la partie la plus complexe et souvent la plus importante pour l'utilisateur. La visualisation de confiance désigne la capacité à représenter correctement à l'utilisateur les différents composants du système avec lesquels il interagit. De plus, les informations véhiculées par le système doivent être convenablement et fidèlement traduites pour qu'un humain puisse en prendre connaissance et les manipuler.

Nous ne nous intéressons pas ici à vérifier que les données représentées sont intelligibles pour l'utilisateur. La correspondance entre représentation et compréhension par l'utilisateur d'une donnée est bien sûr fondamentale, mais elle tient plutôt de l'ergonomie. Nous supposerons de plus que les représentations des informations par les logiciels de confiance sont fonctionnellement correctes. Par conséquent, en l'absence d'intervention d'un attaquant, ce qui est représenté est ce que le logiciel est conçu pour représenter.

Dans ce chapitre nous cherchons à prolonger la portée du contrôle d'accès du système vers l'IHM et son utilisateur, ce dernier étant soucieux de la protection des données auxquelles il accède. Cette problématique est cruciale pour tout système gérant plusieurs domaines de sécurité.

Dans un premier temps, nous cherchons à modéliser les différents composants d'une IHM qui sont pertinents pour l'application d'un contrôle d'accès sur celle-ci. Grâce à cette modélisation, nous définissons notre modèle d'attaquant ainsi que les différentes propriétés de sécurité que nous voulons garantir. Enfin, nous détaillons des stratégies d'implémentation d'IHM qui vérifient les critères formalisés, que nous illustrons avec une évaluation de l'IHM du système Clip OS, décrit en section 5.4.2.

6.1 Modélisation des entités d'une IHM visuelle

Pour la bonne utilisation d'un système sécurisé, il est nécessaire que l'utilisateur soit suffisamment attentif et ait les connaissances requises pour interagir avec l'IHM utilisée. Par ailleurs, notre but n'est pas de protéger l'utilisateur de lui-même. Les connaissances nécessaires regroupent un ensemble de conventions sur l'utilisation de l'environnement de travail et du comportement de cet environnement vis-à-vis de l'interaction utilisateur. Ces connaissances préalables et les informations transmises à l'utilisateur via l'IHM doivent lui permettre de correctement interagir avec les domaines de sécurité. Cette représentation doit être suffisante pour lui permettre de comprendre l'état dans lequel se trouve l'ensemble des applications avec lesquelles il interagit, ainsi que les conséquences potentielles de ses actions.

La connaissance complète de l'état du système manipulé est utopique. Par contre, une représentation des états critiques d'un système par un utilisateur, consciemment ou non, est nécessaire pour utiliser correctement une machine. Afin d'identifier les éléments critiques pour la sécurité, mais également pour une utilisation sûre, nous proposons de définir un ensemble de types d'objets et de sujets.

L'IHM comprend un ensemble d'éléments identifiables par un des sens de l'utilisateur. L'interaction entre l'utilisateur et la machine s'effectue de manière similaire par différents types de capteurs. Il peut s'agir de capteurs de mouvement (p. ex. clavier, souris), de lumière (p. ex. caméra) ou encore de son (p. ex. reconnaissance vocale). Nous nous restreignons ici à l'interface graphique de type écran et aux périphériques d'entrée manipulables physiquement : clavier, souris, *touchpad*, *joystick*, surface tactile... Nous nous appuyons sur les types d'objets du protocole Wayland [53] pour définir des éléments génériques d'IHM.

6.1.1 Surfaces, sorties utilisateur et sorties système

Différents dispositifs matériels peuvent permettre au système de transmettre des informations à l'utilisateur. Nous appelons *sortie système*, notée $o_s \in O_s$, un élément émis par un périphérique physique permettant de transmettre des informations à un utilisateur. Ces éléments sont autant de composantes d'une sortie système que le système peut contrôler de manière indépendante. Dans le cas d'un écran, un pixel est un élément d'une sortie système qui est caractérisé par une taille, une position, une intensité lumineuse, une couleur ou encore une durée d'activation.

Contrairement aux éléments de O_s, une *sortie utilisateur*, notée $o_u \in O_u$, correspond à une réalité perceptible par l'utilisateur, un phénomène qu'il comprend. Pour un affichage graphique, il s'agit d'une part de la partie de l'écran qui est utilisée et d'autre part de l'information qui y est représentée. Dans le cas de l'affichage d'une application, une sortie utilisateur peut correspondre à la fenêtre permettant d'interagir avec l'application ou encore à une zone de saisie de texte contenue dans cette fenêtre. La notion de sortie utilisateur est donc liée à la compréhension de ce qui est transmis à l'utilisateur, via un affichage graphique ou tout autre moyen. Par exemple, le changement d'un pixel entre deux sorties système ne devrait pas présenter une variation suffisante pour que les sorties utilisateur correspondantes soient différentes. De ce point de vue, pour un ensemble de sorties système données, une sortie utilisateur peut donc plus ou moins varier par utilisateur. La bonne interprétation des informations destinées à l'utilisateur est nécessaire pour la bonne utilisation de tout système.

Une *surface* est un espace à deux dimensions dessiné par une application. Il peut s'agir d'une fenêtre ou de l'image d'un curseur. Le compositeur est le sujet responsable d'agréger les surfaces de ses clients et de les exposer sur un ensemble de sorties système. Une surface est caractérisée par une taille, une position relative aux autres surfaces de la même application, des métadonnées comme une icône ou un nom de fenêtre, et un ensemble d'événements comme des frappes clavier qu'elle peut recevoir.

6.1.2 *Seats*, entrées utilisateur et entrées système

Une *entrée utilisateur*, notée $i_u \in I_u$, correspond à l'interprétation par l'utilisateur des entrées qu'il envoie au système. Par exemple, lors de l'utilisation d'un éditeur de texte, lorsque l'utilisateur appuie sur la touche « a » du clavier, il s'attend à ce que l'éditeur reçoive le caractère correspondant et l'affiche. Certaines entrées d'un clavier peuvent correspondre à des combinaisons de touches ou des actions, comme le réglage du volume sonore ou le lancement du navigateur web via une touche dédiée ou un raccourci clavier. Des périphériques de capture comme un microphone ou une caméra peuvent également être utilisés pour interpréter des intentions de l'utilisateur, par exemple par reconnaissance vocale. Ces différentes interactions de l'utilisateur sont regroupées sous l'appellation entrée utilisateur. Les entrées utilisateur qui sont utilisées doivent être connues par convention ou via une sortie utilisateur. Le cas d'un clavier physique avec l'inscription de la signification des touches, associée avec la connaissance du résultat de l'utilisation de ces touches, est un exemple de convention. Autrement dit, il faut que l'utilisateur sache se servir des HID dans son contexte d'utilisation.

Du point de vue du système, une entrée utilisateur est l'abstraction d'une réalité plus riche qu'un simple caractère. Les messages reçus par un HID de type clavier peuvent représenter l'activation de la touche pour une fréquence d'échantillonnage donnée et contiennent notamment l'identifiant de la touche pressée. Une *entrée système*, notée $i_s \in I_s$, correspond à un ensemble de messages, provenant directement ou indirectement d'un HID, reçu par une des couches logicielles qui composent l'interface utilisateur. Suivant la configuration en place,

par exemple la disposition des touches avec leur association à un caractère et à une fenêtre d'observation de leur activation, le système fait la correspondance entre ces informations et une entrée utilisateur. Pour que le système soit correctement utilisé, il est nécessaire que les entrées utilisateur telles que prises en compte par le système soient identiques à celles comprises par l'utilisateur. Pour un clavier, cette interprétation est partiellement assurée par l'inscription sur chaque touche du caractère associé.

Nous prenons également en compte l'utilisation d'entrées indirectes, comme dans le cas de l'utilisation d'un clavier virtuel, grâce à l'invariance des entrées utilisateur. Par exemple, le déclenchement d'une touche virtuelle correspond à un ensemble d'entrées système, typiquement une pression sur un écran tactile, qui est ensuite transformé en un autre ensemble d'entrées système, correspondant ici à une touche du clavier virtuel. Dans ce cas, il y a bien une transformation des entrées système, mais l'entrée utilisateur initiale reste inchangée, car elle correspond toujours à la volonté de l'utilisateur d'envoyer la touche du clavier virtuel. Dans ce cas, comme nous l'avons introduit dans la section 5.2.1, l'originalité des entrées utilisateur est préservée.

Un *seat* identifie l'utilisateur à l'origine d'une entrée utilisateur. Les propriétés d'un *seat* comprennent un ensemble de périphériques comme des HID et des objets virtuels tels qu'un tampon de données utilisé pour le copier-coller. Plusieurs personnes peuvent utiliser une même machine simultanément. Le partage de l'interface entre plusieurs utilisateurs peut être soit physique (machine *multiseat* avec la même session graphique), soit virtuel (partage de l'interface à distance). Le serveur d'affichage différencie les utilisateurs entre eux et leur permet des interactions simultanées avec les applications, dont la sélection et la focalisation de fenêtres graphiques personnalisées, sans que cela ne perturbe l'activité des autres.

6.1.3 Agent utilisateur

Yee [54] définit un *agent utilisateur* (*user-agent*) comme un logiciel dont le but est de servir et de protéger les intérêts de l'utilisateur. Dans notre modèle, un agent utilisateur est un sujet, tel que défini dans la section 2.2.6, capable d'accéder à des entrées et des sorties système. Pour ne pas alourdir inutilement le discours, nous utiliserons simplement *agent* à la place d'agent utilisateur par la suite.

Qu'il s'agisse de *shells* graphiques comme un gestionnaire de fenêtres, de *shells* textuels comme *bash*, ou de navigateurs web, ces applications doivent représenter les intentions de l'utilisateur, ce qui les place donc dans la catégorie des agents. Une interface d'authentification (*greeter*) est un agent qui récupère des entrées utilisateur et affiche sur une sortie utilisateur. Par contre, le service d'authentification qui utilise cette interface peut être un sujet distinct qui ne rentre pas dans la catégorie des agents.

Graphe d'agents utilisateurs

Un agent peut transmettre des entrées utilisateur à d'autres agents qui peuvent à leur tour continuer la transmission. Nous pouvons noter qu'il n'y a pas forcément de garantie d'intégrité sur la donnée initialement transmise. Les agents fils faisant partie de cette hiérarchie peuvent à leur tour transmettre des informations à leur parent pour demander l'affichage d'informations sur une sortie utilisateur.

La figure 6.1 (page 93) expose un exemple simple de hiérarchie d'agents utilisateur pour une session utilisateur. Les données des entrées et sorties utilisateur proviennent des pilotes de périphérique par l'intermédiaire des interfaces exposées par le système. Ces entrées et sorties sont directement accessibles par le compositeur et gestionnaire de fenêtres (*Window Manager* : WM). Pour simplifier, nous supposons qu'un navigateur web est la seule application utilisateur. Elle peut recevoir un sous-ensemble des entrées utilisateur par

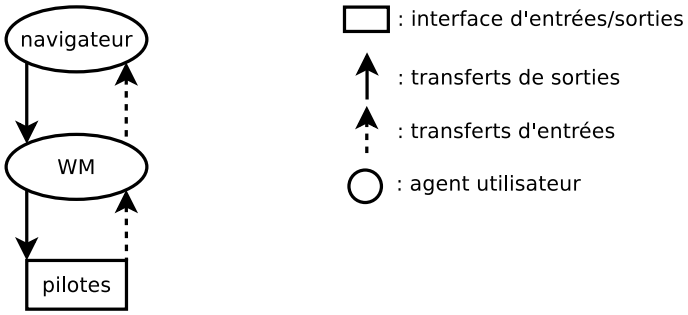


FIGURE 6.1 – Exemple simple de hiérarchie d'agents utilisateur avec un navigateur

l'intermédiaire de son agent parent, ici le gestionnaire de fenêtres. Lorsque le navigateur souhaite afficher des informations sur des sorties utilisateur, il en fait la demande à son agent parent qui est alors chargé de les transmettre au pilote d'affichage.

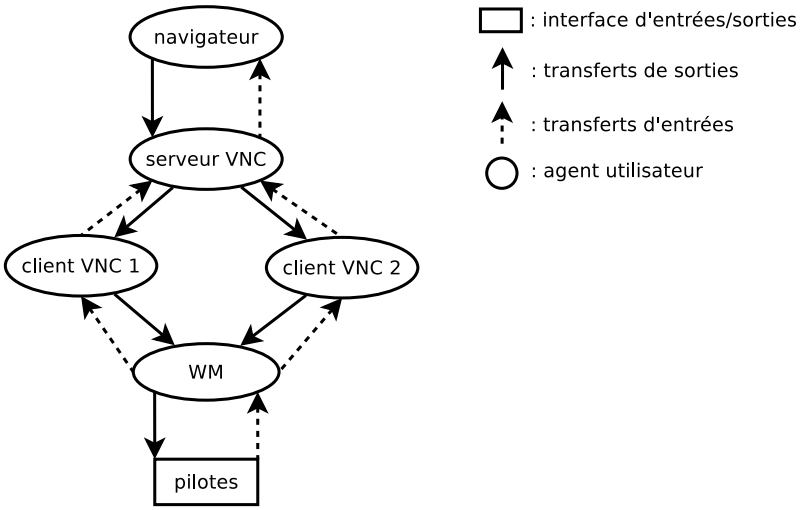


FIGURE 6.2 – Exemple complexe de hiérarchie d'agents utilisateur avec deux clients VNC et un navigateur

La figure 6.2 représente un cas peu commun, mais réaliste, d'utilisation d'un service de déport d'affichage. Il permet de justifier que les relations entre agents se modélisent non pas avec un simple arbre, mais bien avec un graphe. Il s'agit ici de connexions simultanées de deux clients, à partir d'une même session utilisateur, à un service donné. Ce service est ici représenté par VNC, mais il est possible d'envisager d'autres cas d'usage comme une connexion SSH vers une session d'un multiplexeur de terminal comme *tmux*. Le scénario de l'exemple met en situation l'utilisateur qui ouvre un premier client de déport d'affichage VNC pour se connecter sur un compositeur graphique d'une session distante. Dans cette session graphique distante, un navigateur web est lancé. L'utilisateur lance ensuite un deuxième client VNC pour se connecter sur la même session distante que le premier client VNC. Du point de vue du gestionnaire de fenêtres local, il y a un seul utilisateur connecté, représenté par un seul *seat*. Concernant le compositeur distant, mis à disposition par un service VNC, deux clients VNC y sont connectés, donc potentiellement deux utilisateurs différents, représentés par deux *seats* distincts. Le graphe des agents peut évoluer au cours du temps, en raison de déconnexions ou de terminaisons. Par exemple, si les clients VNC se déconnectent du serveur VNC, le compositeur VNC et le navigateur peuvent continuer à s'exécuter, mais ne sont plus des agents du point de vue du *seat* local utilisé.

Cet exemple d'utilisation permet également de noter que les communications entre agents

peuvent consister en des protocoles et canaux de communication différents. Dans le cas d'un système basé sur Linux, le compositeur local lit les informations des périphériques d'entrée via les pseudo-fichiers situés dans `/dev/input/`, selon le protocole défini par les pilotes. Les communications entre le compositeur local et ses clients, les deux clients VNC pour la figure 6.2, s'effectuent via un protocole dédié comme Wayland. Enfin, les communications entre les clients VNC et le serveur VNC s'effectuent via le protocole VNC.

Agent racine

Nous définissons un *agent racine* comme un agent qui a un accès direct aux interfaces de sortie et d'entrée associées à un *seat*. Dans la représentation sous forme de graphe, les agents racines correspondent aux nœuds formant les sources des entrées et les destinations des sorties utilisateur.

Par la suite, nous excluons volontairement toute simulation des entrées utilisateur par un domaine autre qu'un domaine racine. Dans notre modèle, une prise en main à distance ne simule pas un utilisateur, mais représente un autre utilisateur du système, qui a accès à un sous-ensemble des sorties système, mais opère à partir d'un *seat* qui lui est dédié.

La figure 5.1 permet d'identifier le compositeur et gestionnaire de fenêtres comme agent racine étant donné son accès direct à l'interface du périphérique matériel. Dans le cas de la figure 6.2 où il y a plusieurs compositeurs, seul le compositeur local est l'agent racine. Le compositeur VNC n'est pas considéré comme un agent racine pour le *seat* de l'exemple, car il ne représente ici qu'un maillon au milieu d'autres agents.

Entrées utilisateur déterminées ou indéterminées et agent focalisé

Nous supposons que la signification des entrées utilisateur est connue de l'utilisateur par convention ou configuration, de l'une des manières suivantes. La première manière de faire connaître la signification des entrées utilisateur est par l'intermédiaire d'une information sur des éléments physiques de l'IHM, par exemple l'inscription de lettres sur les touches de clavier. Le deuxième moyen de connaître la signification des entrées utilisateur est via les sorties utilisateur, comme c'est le cas pour un clavier virtuel. Le cas échéant, il sera alors nécessaire d'avoir un moyen de valider l'information transmise par ces sorties utilisateur.

Nous pouvons classer les entrées utilisateur en deux catégories : les *entrées déterminées* et les *entrées indéterminées*. Des entrées sont dites déterminées si elles sont assignées à un agent utilisateur défini a priori. Ces entrées ainsi que les agents associés sont supposés connus de l'utilisateur, soit par convention, soit par configuration. Cette configuration est dépendante des agents, particulièrement de l'agent racine, autrement dit de la configuration du système.

Nous pouvons citer comme exemple les combinaisons de touches clavier (`alt+tab`) qui permettent de passer d'une application à l'autre. Dans ce cas, c'est le gestionnaire de fenêtres, correspondant habituellement à un agent racine, qui reçoit ces entrées. Dans le cas des touches de commande multimédia, il peut s'agir d'un autre agent, faisant partie d'une application multimédia.

Les entrées qui ne font pas partie des entrées déterminées sont appelées entrées indéterminées. Ces entrées indéterminées ne sont pas assignées à un agent particulier, mais dépendent de l'état du système. Un *agent focalisé* est défini comme l'agent qui reçoit toutes les entrées indéterminées d'un *seat* donné sans les retransmettre.

Par exemple, les touches alphanumériques d'un clavier sont par convention un sous-ensemble des entrées indéterminées. L'utilisateur peut indiquer au gestionnaire de fenêtres de changer l'agent qui va recevoir ces entrées, habituellement soit par une sélection de fenêtres, soit par un raccourci clavier.

Un agent focalisé est identifié comme tel par son agent parent à un instant donné, ce qui permet à l'utilisateur d'en être averti. Cette information est habituellement affichée à l'utilisateur par l'agent parent. Dans le cas d'un gestionnaire de fenêtres, cela se traduit souvent par un changement de couleur de la barre de titre associée à l'agent focalisé si celui-ci est une application. Pour les systèmes limités par la taille de l'affichage, comme c'est souvent le cas des smartphones, l'application focalisée est habituellement l'application qui utilise la majorité de l'affichage.

Il n'est pas nécessaire que l'association faite par l'utilisateur entre les entrées et leur destinataire soit une fonction totale de toutes les combinaisons d'entrées, mais seulement une fonction totale des entrées que l'utilisateur va utiliser. Nous supposons que l'utilisateur connaît le comportement des entrées qu'il souhaite utiliser et l'agent qui doit les recevoir. Dans le cas contraire, il n'est pas possible de raisonner sur ces entrées.

6.1.4 Domaines système et domaines utilisateur

Comme nous l'avons défini dans la section 2.2.5, un domaine de sécurité permet de restreindre l'ensemble des objets accessibles par les sujets appartenant au domaine. Dans ce chapitre nous nommons *domaine système* un domaine de sécurité du point de vue du système. Contrairement aux chapitres précédents, nous nommons *domaine utilisateur* une représentation d'un ensemble de domaines de sécurité que le système retranscrit à l'utilisateur. Cette vue abstraite doit permettre à l'utilisateur d'identifier des cercles de diffusion d'information sur le système qu'il utilise.

Dans notre modèle, un domaine utilisateur n'est pas forcément ce que pense l'utilisateur, mais ce que le concepteur de l'IHM de confiance anticipe comme compris par l'utilisateur.

6.2 Propriétés de sécurité d'une IHM

La problématique de modélisation d'une interface homme-machine repose sur la représentation mentale que l'utilisateur se fait de la machine, et plus particulièrement de l'état du système. L'utilisateur possède une vision du système réduite à un sous-ensemble de l'architecture réelle du système qui est beaucoup plus complexe. Afin d'être capable d'exprimer des propriétés sur une IHM, il est nécessaire de modéliser la représentation du fonctionnement du système que se construit l'utilisateur. Alors que le problème est déjà complexe dans le cas d'un système, la modélisation de tous les aspects du comportement de l'utilisateur est hors de portée. Nous nous attachons ici à formaliser l'interaction de l'utilisateur avec une machine sous le seul angle de la sécurité. Lorsque l'utilisateur interagit avec une IHM, il peut générer des entrées utilisateur en fonction de sa représentation de l'état du système. En retour, il attend une conséquence de son action qui comprend notamment des informations, via des sorties utilisateur, lui permettant d'actualiser sa représentation du système.

Notre point de vue est celui du *concepteur d'une IHM de confiance*. Une *abstraction* correspond à ce qu'il veut que l'utilisateur comprenne du système. Notre hypothèse de base suppose que l'abstraction du système est bien comprise par l'utilisateur, ce qui peut être validé par des études ergonomiques.

6.2.1 Formalisation du modèle de l'interaction utilisateur avec la machine

Automate de représentation du système

Un état, noté $\eta_s \in H_s$ pour le système et $\eta_u \in H_u$ pour l'utilisateur, correspond au graphe des agents, respectivement du point de vue du système et respectivement du point de vue de l'utilisateur à un instant donné.

Nous formalisons l'évolution du système et de sa représentation par l'utilisateur par un automate. Un état de cet automate correspond au couple d'un état système et d'un état utilisateur : (η_s, η_u). L'ensemble des états est noté $\Sigma = H_s \times H_u$. Une exécution du système et sa représentation par l'utilisateur correspondent à une trace dans l'automate. Nous modélisons ainsi la compréhension et l'influence de l'utilisateur sur l'état du système.

Un événement système, noté $e_s \in E_s$, représente tout événement ne provenant pas de l'utilisateur, mais qui peut uniquement avoir une influence sur η_s. Ces événements peuvent représenter la création de nouveaux agents ou encore la terminaison d'agents existants. Un événement peut provenir de requêtes système comme les IPC, RPC ou tout autre type de signal qui peut avoir un impact sur un agent. Par exemple, une tâche programmée pour s'exécuter à un instant particulier peut modifier l'état système sans intervention de l'utilisateur.

Comme vu en section 6.1.3, à chaque entrée utilisateur i_u, par exemple l'utilisation d'un raccourci clavier ou la saisie d'un caractère, correspond un ensemble de messages appelé entrée système i_s.

Pour exprimer l'équivalence entre une notion système et sa représentation par l'utilisateur, telle qu'anticipée par le concepteur de l'IHM, nous utilisons une fonction d'abstraction notée $\overline{\cdot}$. L'abstraction $\overline{i_s}$ correspond à une entrée i_u, et l'abstraction $\overline{o_s}$ correspond à une sortie o_u. Par extension, $\overline{I_s}$ correspond à l'abstraction de l'ensemble des entrées système et $\overline{O_s}$ correspond à l'abstraction de l'ensemble des sorties système.

Une transition d'un état de l'automate à un autre est étiquetée par un événement système, une abstraction d'une entrée système, ou une abstraction d'une sortie système. L'ensemble des étiquettes est noté $L = E_s \cup \overline{I_s} \cup \overline{O_s}$.

Suivant la notation 1 (page 27), l'automate de représentation des états système et des états utilisateur est défini sous forme d'un LTS $\mathcal{A} = \langle \Sigma, L, \rightarrow \rangle$. Un événement système e_s modifie un état système η_s, ce qui implique une transition de l'automate de (η_s, η_u) vers (η'_s, η_u). La fonction de transition notée $\xrightarrow{e_s}$ est définie par :

$$(\eta_s, \eta_u) \xrightarrow{e_s} (\eta'_s, \eta_u)$$

De même, une entrée système i_s provenant de l'utilisateur a un impact sur l'état système η_s ainsi que sur l'état utilisateur η_u. Par exemple, un utilisateur qui actionne un bouton attend un changement d'état du système, comme la fermeture d'une fenêtre ou encore l'affichage d'un caractère correspondant à la touche sur laquelle il vient d'appuyer. Cette attente de l'utilisateur, relative à une de ses entrées, et la prise en compte de cette entrée par le système, correspond à une transition formalisée par :

$$(\eta_s, \eta_u) \xrightarrow{\overline{i_s}} (\eta'_s, \eta'_u)$$

Enfin, une sortie système o_s prise en compte par l'utilisateur a un impact sur son état. Par exemple, l'affichage d'une fenêtre informe l'utilisateur de l'application focalisée, qui va donc recevoir les prochaines entrées :

$$(\eta_s, \eta_u) \xrightarrow{\overline{o_s}} (\eta_s, \eta'_u)$$

Notre modélisation s'intéresse à la sortie prise en compte par l'utilisateur, ce qui n'a pas d'influence directe sur l'état du système. Les changements d'état du système qui peuvent survenir suite à une sortie o_s sont représentés avec la transition $\xrightarrow{e_s}$.

Nous prenons ici en compte les actions dont l'utilisateur a conscience, ce qui représente également les limites de notre modèle. Nous ignorons volontairement les événements externes au système qui pourraient avoir une influence sur la compréhension du système par l'utilisateur. En effet, avec une bonne compréhension du système ou des informations supplémentaires provenant de son environnement, l'utilisateur peut prévoir un état sans attendre de le constater. C'est par exemple le cas lors de l'anticipation d'un événement comme une tâche programmée ou l'actualisation d'une page web.

Association des agents et des domaines

La notion d'agent peut être différente pour le système et l'utilisateur. Le système représente graphiquement une abstraction de ses agents à l'utilisateur. L'abstraction d'un agent système a_s par un agent utilisateur a_u est notée $\overline{a_s}$. Chaque agent a_u que le concepteur représente à l'utilisateur correspond à l'abstraction d'un agent système. Plusieurs agents système peuvent avoir la même abstraction, mais tout agent système a une abstraction comprise par l'utilisateur :

$$\forall a_s \in A_s, \exists a_u \in A_u, a_u = \overline{a_s}$$

Un domaine système, noté $d_s \in D_s$, est associé à chaque agent système. Les agents non contraints sont assignés à un domaine par défaut. L'identification du domaine d'un agent est donnée par la fonction totale suivante :

$$\textsc{Domain} : A_s \longrightarrow D_s$$

De la même manière que le système associe un agent système à un domaine système, l'utilisateur peut associer un agent utilisateur à son domaine utilisateur, noté $d_u \in D_u$. Cette interprétation de l'utilisateur est donnée par la fonction totale suivante :

$$\overline{\textsc{Domain}} : A_u \longrightarrow D_u$$

Chaque domaine système d_s tel que pris en compte par le système correspond à un domaine utilisateur $\overline{d_s}$, destiné à être interprété par l'utilisateur. Un domaine utilisateur est un ensemble de domaines système voués à être regroupés dans la vue de l'utilisateur. Selon notre approche, cette abstraction correspond à ce que l'IHM du système est conçue pour représenter. Elle ne dépend pas de l'utilisateur, que nous considérons apte à correctement comprendre ce qui lui est représenté.

Les ensembles de domaines système formés par l'abstraction ainsi définie vérifient l'hypothèse suivante. L'abstraction du domaine système d'un agent système est identique au domaine utilisateur de l'abstraction du même agent système. Plus formellement :

$$\forall a_s \in A_s, \overline{\textsc{Domain}(a_s)} = \overline{\textsc{Domain}}(\overline{a_s})$$

Par la suite nous écrivons qu'un domaine peut recevoir des entrées ou afficher sur des sorties lorsqu'un des agents qui le compose en est capable. De même qu'il y a une hiérarchie pour les agents, celle-ci est transposée pour leurs domaines.

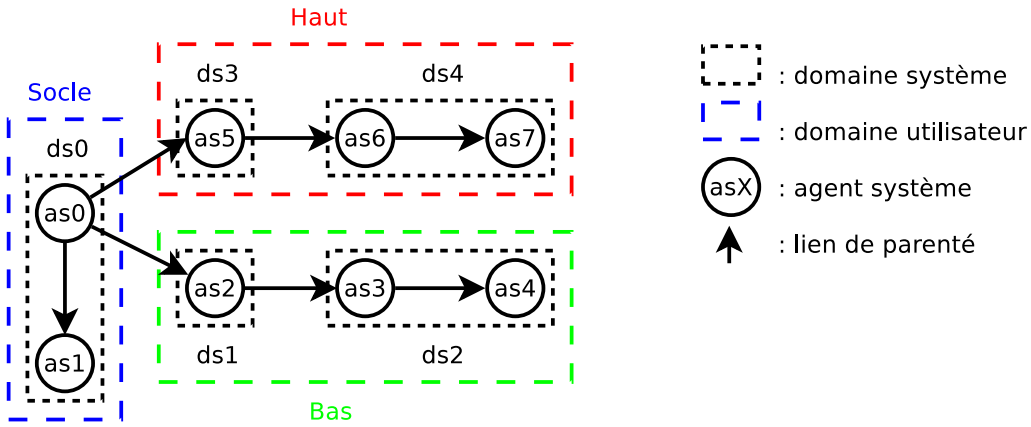


FIGURE 6.3 – Graphe d'agents et de domaines pour CLIP OS du point de vue du système

6.2.2 Modèle de CLIP OS

Comme détaillé dans la section 5.4.2, CLIP OS est un système capable de gérer plusieurs niveaux de sensibilité. La figure 6.3 représente un sous-ensemble d'agents et de quelques domaines pertinents pour un poste utilisateur CLIP OS. Trois domaines utilisateur lui sont exposés : les domaines *Socle* (bleu), *Bas* (vert) et *Haut* (rouge). Le domaine *Socle* fait référence à l'environnement directement accessible après authentification de l'utilisateur. Ce domaine permet la configuration du système ainsi que la navigation entre les domaines *Bas* et *Haut*. Les domaines *Bas* et *Haut* mettent chacun à disposition un espace de travail pour l'utilisateur. L'espace de travail *Bas* est directement relié à Internet alors que l'espace de travail *Haut* est relié à un réseau privé virtuel.

Le domaine utilisateur *Socle* comprend deux agents. L'agent racine a_s^0 est le compositeur chargé d'afficher les fenêtres du socle et de sélectionner le domaine *Bas* ou *Haut*. L'agent a_s^1 correspond à une barre de menu permettant d'accéder à divers outils. Ces deux agents sont dans un domaine système d_s^0. Le domaine utilisateur *Socle* coïncide avec $\overline{d_s^0}$.

Le domaine utilisateur *Bas* comprend trois agents. L'agent a_s^2 est un client VNC qui se connecte à un serveur VNC représenté par l'agent a_s^3. Pour simplifier cet exemple, nous supposons que l'agent a_s^3 représente aussi le bureau de l'environnement *Bas*. Cet enchaînement permet de joindre l'agent a_s^4 qui représente un navigateur web.

Concernant les domaines système, l'agent a_s^2 est dans un domaine système de sécurité d_s^1 alors que l'agent a_s^3 est dans un autre domaine système d_s^2. Le domaine utilisateur *Bas* est l'abstraction de d_s^1 et d_s^2.

Le domaine utilisateur *Haut* comprend également trois agents, suivant la même logique que le domaine *Bas*. L'agent a_s^5 est un client VNC qui se connecte à un serveur VNC représenté par l'agent a_s^6, représentant également le bureau de l'environnement *Haut*. L'agent a_s^7 représente une autre application utilisateur.

La figure 6.4, abstraction de la figure 6.3, représente l'abstraction utilisateur présentée par le système. Les agents a_u^0 et a_u^1 sont représentés dans la vue système et utilisateur : $a_u^0 = \overline{a_s^0}$ et $a_u^1 = \overline{a_s^1}$. Par contre, l'utilisateur n'a pas conscience des agents a_s^3 et a_s^2 qui sont la partie VNC de CLIP OS, volontairement transparente pour l'utilisateur, mais il les interprète comme l'agent du bureau : $a_u^2 = \overline{a_s^3} = \overline{a_s^2}$. Le navigateur web est interprété par l'utilisateur comme l'agent $a_u^3 = \overline{a_s^4}$. L'abstraction se décline pour le niveau *Haut* de manière similaire.

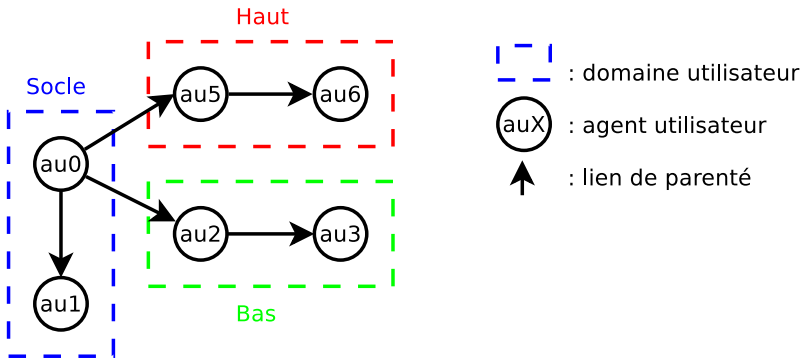


FIGURE 6.4 – Graphe d'agents et de domaines pour CLIP OS du point de vue de l'utilisateur

6.2.3 Propriétés d'une IHM de confiance

Une interface homme-machine de confiance doit protéger les échanges entre l'utilisateur et les domaines de sécurité, en intégrité et en confidentialité. Après une définition du modèle d'attaquant, nous formalisons les propriétés de sécurité à vérifier pour garantir cette protection.

Modèle d'attaquant et domaine de confiance

Beckert et Beuster [55] décrivent l'intégrité et la confidentialité du point de vue du système, c'est-à-dire pour la protection des données connues et manipulées par celui-ci. D'après leur définition de la confidentialité, aucun secret ne doit fuir par l'interface utilisateur. L'utilisateur est donc considéré comme un attaquant potentiel pour le système. Notre modèle d'attaquant est différent ; il consiste à protéger l'utilisateur contre des sujets malveillants correspondant à des domaines de sécurité.

Beckert et Beuster [55] formalisent des propriétés de sécurité dans lesquelles ils considèrent le système comme un seul sujet également agent utilisateur. Nous nous intéressons ici à modéliser un système réel comprenant de multiples agents, liés à de multiples domaines de sécurité, qui se partagent l'IHM.

Dans notre formalisation, la menace est constituée par le comportement possiblement malveillant d'un agent, comme l'usurpation de l'affichage ou encore la récupération frauduleuse d'informations via l'IHM. La compromission d'un agent signifie qu'il peut compromettre les autres agents de son domaine. Un domaine système est donc considéré comme entièrement sain ou entièrement compromis.

L'attaquant n'est pas supposé capable de compromettre n'importe quel domaine. Concernant l'IHM, un domaine (de confiance) fait partie de la TCB lorsque tous ses agents :

— sont considérés comme non corruptibles par le modèle de sécurité du système ;
— et lorsqu'ils sont fonctionnellement corrects, ce qui signifie qu'ils exposent de manière intelligible pour l'utilisateur les sorties système et interprètent correctement les entrées utilisateur.

Nous considérons donc que l'attaquant est capable de compromettre n'importe quel agent d'un domaine qui ne fait pas partie de la TCB. Ceci ne signifie pas pour autant que tous les domaines qui ne font pas partie de la TCB sont compromis. Notre modèle s'attache à identifier l'impact de la compromission d'un ensemble de domaines sur les domaines qui restent sains.

En ce qui concerne les domaines utilisateur, nous écrivons qu'un domaine utilisateur est de confiance (respectivement sain) si tous les domaines système dont il constitue l'abstraction sont de confiance (respectivement sains). Un domaine utilisateur est compromis dès que

l'un des domaines système qu'il abstrait est compromis.

Dans le cas de Clip OS, le domaine *Socle* fait partie de la TCB. Par contre, les domaines *Bas* et *Haut* sont considérés comme potentiellement compromis, car pouvant contenir des agents malveillants.

Présentation des propriétés attendues

Les critères de sécurité expliqués dans la section 5.2 permettent ici de définir des propriétés sur les domaines de notre modèle.

La préservation de l'accomplissement des entrées nécessite que les informations transmises par l'utilisateur à un domaine cible arrivent au moins jusqu'à ce domaine avec une abstraction identique à celle de l'entrée utilisateur initiale.

L'originalité des entrées utilisateur, correspondant à un *seat*, permet à un domaine d'identifier de manière fiable la provenance de commandes utilisateur. L'originalité est une hypothèse de base qui permet de définir les entrées utilisateur.

Pour se protéger contre des attaques de type *confused deputy*, l'utilisateur doit être en mesure d'attribuer les sorties utilisateur aux domaines utilisateur qui en sont à l'origine.

Par définition, tous les domaines de notre modèle sont potentiellement autorisés à afficher leurs sorties. Cette problématique fonctionnelle est soumise à des règles d'utilisation dépendantes de l'implémentation de l'IHM, ce qui sera abordé dans la section 6.3.1.

Enfin, pour assurer la confidentialité des échanges via l'IHM, il faut d'une part que les données saisies par l'utilisateur ne soient accessibles au plus qu'aux domaines auxquels l'utilisateur pense envoyer ces informations. Ceci inclut le domaine racine ainsi que tous les domaines qui voient légitimement transiter les données jusqu'au domaine cible. D'autre part, la préservation de la confidentialité nécessite que les sorties système ne soient connues qu'au plus par les domaines dont l'utilisateur pense qu'ils ont accès à ces informations. Ceci inclut le domaine source et les domaines qui voient légitimement transiter les données pour qu'elles puissent être présentées à l'utilisateur.

Accomplissement et confidentialité des entrées

Comme détaillé dans la section 5.2.1, les entrées sont sujettes à des attaques impactant leur accomplissement et leur confidentialité. Mettre à mal l'accomplissement d'une entrée consiste tout simplement à bloquer sa transmission vers le domaine utilisateur cible. Afin d'identifier les domaines qui reçoivent les entrées utilisateur, nous avons besoin d'identifier l'ensemble des domaines auxquels les entrées utilisateur sont transmises.

La fonction Receiver associe à un état système et une entrée système donnés les domaines système qui reçoivent successivement cette entrée :

$$\textsc{Receiver} : H_s \times I_s \longrightarrow \wp(D_s)$$

Il est important de souligner que les domaines système peuvent transformer une entrée système en une autre avant de la transmettre. Ceci peut se produire lors d'une rupture protocolaire entre agents. C'est bien l'ensemble des domaines qui participent au traitement de l'entrée système, dont la valeur initiale est $i_s \in I_s$, qui est décrit par Receiver.

Son homologue dans le monde utilisateur est $\overline{\textsc{Receiver}}$. Elle associe les domaines utilisateur qui reçoivent, à un état utilisateur et une entrée utilisateur donnés :

$$\overline{\textsc{Receiver}} : H_u \times I_u \longrightarrow \wp(D_u)$$

Propriété 1 (Accomplissement d'entrée)**.** *L'accomplissement de l'entrée utilisateur $\overline{i_s}$ est assuré si et seulement si les domaines utilisateur qui reçoivent cette entrée correspondent à*

un sous-ensemble des abstractions des domaines système qui reçoivent réellement une entrée système de sémantique équivalente :

$$\overline{\textsc{Receiver}}(\eta_u, \overline{i_s}) \subseteq \overline{\textsc{Receiver}(\eta_s, i_s)}$$

Contrairement à l'accomplissement qui doit être vérifié sur les entrées utilisateur, la confidentialité doit être vérifiée sur les entrées système. En effet, nous ne nous intéressons pas à protéger directement la confidentialité de l'abstraction des données qui transitent sur le système, mais bien celle des données brutes qui permettent d'inférer cette abstraction.

Propriété 2 (Confidentialité d'entrée)**.** *La confidentialité de l'entrée système i_s est assurée si et seulement si les domaines anticipés par l'utilisateur comme récipiendaires potentiels de l'abstraction de cette entrée correspondent à un sur-ensemble des abstractions des domaines système qui reçoivent réellement cette entrée :*

$$\overline{\textsc{Receiver}(\eta_s, i_s)} \subseteq \overline{\textsc{Receiver}}(\eta_u, \overline{i_s})$$

Attribution et confidentialité des sorties

Telle que définie dans la section 5.2.2, l'attribution d'une sortie utilisateur aux domaines utilisateur qui en sont à l'origine correspond à l'identification fiable des sources d'une donnée retranscrite sur un affichage. En complément, comme pour les entrées utilisateur, les sorties peuvent être lues par différents domaines, ce qui a un impact sur la confidentialité des données affichées.

La fonction Displayer associe à un état système et une sortie système donnés les domaines système qui exposent successivement cette sortie :

$$\textsc{Displayer} : H_s \times O_s \longrightarrow \wp(D_s)$$

Son homologue dans le monde utilisateur est $\overline{\textsc{Displayer}}$. Elle associe les domaines utilisateur qui exposent, à un état utilisateur et une sortie utilisateur donnés :

$$\overline{\textsc{Displayer}} : H_u \times O_u \longrightarrow \wp(D_u)$$

Propriété 3 (Attribution de sortie)**.** *L'attribution d'une sortie utilisateur $\overline{o_s}$ est assurée si et seulement si les domaines utilisateur qui exposent cette sortie correspondent aux domaines système qui exposent réellement une sortie système de sémantique équivalente :*

$$\overline{\textsc{Displayer}}(\eta_u, \overline{o_s}) = \overline{\textsc{Displayer}(\eta_s, o_s)}$$

Lors d'une lecture de sortie système, par exemple pour une capture d'écran, un domaine système est autorisé à accéder à un sous-ensemble des sorties utilisateur d'un autre domaine. Une capture complète des écrans nécessite d'accéder à la totalité des sorties du domaine racine. Par contre, une capture de seulement une fenêtre d'application ne nécessite d'accéder qu'aux sorties du domaine correspondant. Nous pouvons noter qu'un domaine parent a besoin d'accéder aux sorties de ses fils s'il est responsable de l'affichage de ces sorties. Cependant, s'il n'est responsable que de la composition de ces sorties, mais pas directement de leur affichage, il est possible de conserver la confidentialité d'une sortie en n'effectuant qu'un transfert d'une référence de cette sortie et non une retranscription de celle-ci. Un exemple concret sera détaillé dans la section 6.4.1.

La fonction Watcher associe à un état système et une sortie système donnés les domaines système qui observent successivement cette sortie :

$$\textsc{Watcher} : H_s \times O_s \longrightarrow \wp(D_s)$$

Son homologue dans le monde utilisateur est $\overline{\text{WATCHER}}$. Elle associe les domaines utilisateur qui observent, à un état utilisateur et une sortie utilisateur donnés :

$$\overline{\text{WATCHER}} : H_u \times O_u \longrightarrow \wp(D_u)$$

Contrairement à l'attribution qui doit être vérifiée sur les sorties utilisateur, la confidentialité doit être vérifiée sur les sorties système. En effet, contrairement à Beckert et Beuster, nous considérons que les données qui doivent être affichées sont de fait destinées à l'utilisateur. Nous nous attachons donc à protéger les données qui transitent sur le système. Implicitement, la protection d'une sortie système concerne également son abstraction.

Propriété 4 (Confidentialité de sortie)**.** *La confidentialité d'une sortie système o_s est assurée si et seulement si les domaines anticipés par l'utilisateur comme étant capables d'observer l'abstraction de cette sortie correspondent à un sur-ensemble des abstractions des domaines système qui observent réellement cette sortie :*

$$\overline{\text{WATCHER}(\eta_s, o_s)} \subseteq \overline{\text{WATCHER}}(\eta_u, \overline{o_s})$$

Exécution sécurisée

Chaque état doit permettre à l'utilisateur d'avoir la connaissance du résultat de l'application des fonctions $\overline{\text{RECEIVER}}$, $\overline{\text{DISPLAYER}}$ et $\overline{\text{WATCHER}}$. Une exécution, formalisée par une trace, est dite sécurisée si elle vérifie les propriétés suivantes pour chaque transition :

— préalablement à une transition engendrée par une entrée, les propriétés de sécurité sur l'entrée assurant leur accomplissement et protection en confidentialité sont vérifiées par le couple d'états système et utilisateur ;
— consécutivement à une transition engendrée par une sortie, les propriétés de sécurité sur la sortie assurant leur protection en confidentialité et leur attribution sont vraies pour le couple d'états système et utilisateur.

Plus formellement, une exécution est dite sécurisée si et seulement si :

— chacune de ses transitions de la forme $(\eta_s, \eta_u) \xrightarrow{\overline{i_s}} (\eta'_s, \eta'_u)$ vérifie les propriétés 1 et 2 pour les états source, ce qui correspond à $\overline{\text{RECEIVER}}(\eta_u, \overline{i_s}) = \overline{\text{RECEIVER}}(\eta_s, i_s)$;
— et chacune de ses transitions de la forme $(\eta_s, \eta_u) \xrightarrow{\overline{o_s}} (\eta_s, \eta'_u)$ vérifie les propriétés 3 et 4 pour les états destination, c'est-à-dire $\overline{\text{DISPLAYER}}(\eta'_u, \overline{o_s}) = \overline{\text{DISPLAYER}}(\eta_s, o_s)$ et $\overline{\text{WATCHER}(\eta_s, o_s)} \subseteq \overline{\text{WATCHER}}(\eta'_u, \overline{o_s})$.

Cette définition impose donc une synchronisation entre la vue utilisateur et la vue système tout au long d'une exécution, de telle sorte que l'utilisateur ait connaissance des éléments critiques qu'il voit et manipule, et que ces informations soient et restent correctes vis-à-vis du système.

6.3 Mise en place d'une IHM de confiance

Tout d'abord, afin de vérifier les propriétés de sécurité que nous avons définies, seules une entrée utilisateur ou une sortie utilisateur peuvent avoir une influence sur la compréhension de l'utilisateur du système. Comme l'automate de représentation du système le définit, un événement système e_s ne peut pas influer sur l'état utilisateur η_u.

6.3.1 Affichage et attribution des sorties

L'affichage des sorties d'un domaine peut impacter la disponibilité ou l'utilisabilité d'une IHM, mais également la capacité de l'utilisateur à attribuer une sortie. Dans un premier temps, le contrôle d'accès du système est garant du fait que seuls les agents sont autorisés à

communiquer, directement ou indirectement, avec un agent racine. Le contrôle d'accès peut alors affiner les autorisations par régions d'affichage sur les périphériques concernés. Une première façon est tout simplement d'avoir recours à un découpage statique de l'affichage en régions et de les attribuer à un ensemble de domaines. Cela peut consister à avoir un écran physique dédié à chaque domaine, ou encore à adopter une convention géographique de répartition des domaines sur un écran (*region tiling* [71]). Cette approche est simple à implémenter, mais induit une limitation artificielle de l'espace disponible pour l'affichage.

L'approche la plus commune est de déléguer à l'utilisateur le contrôle d'accès via un gestionnaire de fenêtre qui lui est dédié. En effet, le positionnement et le dimensionnement d'une fenêtre définissent une région sur laquelle un agent peut afficher. Pour vérifier nos propriétés de sécurité, il est cependant nécessaire de faire connaître à l'utilisateur, de manière dynamique, l'attribution des sorties utilisateur.

L'attribution des sorties consiste à vérifier la propriété 3 pour l'état (η_s, η'_u). Les domaines d'où provient une partie donnée de l'affichage peuvent être connus soit par convention, comme c'est le cas pour un découpage statique de l'affichage, soit par une sortie utilisateur de confiance, elle-même transmise via l'IHM. Garantir la fiabilité de cette information impose des contraintes de sécurité sur l'IHM. Différentes implémentations permettent d'exposer cette information de manière fiable.

Pour apporter cette notion d'attribution des sorties utilisateur, nous avons vu dans la section 5.3.2 qu'une solution consiste à définir un chemin de confiance. Nous définissons une *zone de confiance* comme l'implémentation d'un chemin de confiance pour une interface graphique. Une zone de confiance est une sortie utilisateur qui doit contenir des informations fidèles à la réalité système, compréhensibles par l'utilisateur. Ces informations doivent être connues de manière sûre par un domaine système de confiance, qui se charge de l'afficher via une sortie utilisateur que seuls des domaines utilisateur de confiance peuvent modifier. Afin de se protéger contre des attaques de type *picture-in-picture*, décrites dans la section 5.2.2, une zone de confiance doit faire état de l'attribution des sorties utilisateur. Chaque domaine utilisateur en capacité de modifier une sortie utilisateur, que ces domaines soient malveillants ou non, doit donc être connu de l'utilisateur. Ce dernier doit également connaître au préalable l'emplacement de cette zone de confiance et qu'il la prenne en compte tout au long de l'utilisation de l'IHM.

La capacité d'un domaine à attribuer une sortie utilisateur peut être garantie de trois manières : identification assurée par la TCB, authentification cryptographique ou secret partagé. L'identification de la source d'un message reçu par un domaine de confiance peut être implémentée avec un système d'IPC fournissant une fonctionnalité d'identification de confiance de l'émetteur, comme le permettent les sockets UNIX avec des messages `SCM_CREDENTIALS`. Plus simplement, la mise à disposition de canaux de communication dédiés à la communication entre deux domaines, ce qui est assuré par le contrôle d'accès, peut également permettre d'identifier l'émetteur.

La deuxième solution permettant l'attribution de l'origine d'une sortie utilisateur à un domaine est l'utilisation de signature cryptographique des données. Comme pour la propriété de confidentialité, la sécurité de cette méthode repose sur la capacité à garder secrètes les clefs privées. En cas de compromission du domaine, un attaquant pourrait usurper l'identité des domaines dont il a la maîtrise.

Ces deux premières approches d'attribution des sorties utilisateur sont complémentaires. L'approche par IPC simplifie l'attribution locale tandis que l'approche cryptographique est particulièrement importante pour des communications réseau en présence d'un attaquant.

Une troisième méthode d'attribution consiste à partager un secret entre le domaine source d'une sortie et l'utilisateur. Cette technique, dite du *petname*, est détaillée dans la section 5.3.4. Elle peut permettre à l'utilisateur d'attribuer une sortie utilisateur à un

domaine utilisateur dans le cas où les sorties sont exclusivement inscriptibles par un seul domaine.

6.3.2 Confidentialité des entrées et des sorties

Un HID virtuel, comme un clavier virtuel, qui est affiché à l'utilisateur ne doit être considéré comme une source d'entrée utilisateur que si le domaine qui affiche ce périphérique à l'utilisateur et transforme les entrées système est un domaine de confiance et identifié comme tel par l'utilisateur.

Le respect de la confidentialité des transmissions d'entrées système ou de sorties système peut être assuré soit grâce aux propriétés de confidentialité d'une IPC fournie par la TCB, soit par un chiffrement cryptographique. Par exemple, l'utilisation de socket UNIX avec un contrôle d'accès correctement configuré peut permettre de conserver la confidentialité d'échanges entre des processus. Les serveurs d'affichage *X.Org* et Weston (implémentation du protocole Wayland) peuvent utiliser de tels sockets pour communiquer avec leurs agents. L'utilisation du chiffrement asymétrique requiert l'utilisation d'une autorité de certification et la mise à disposition de certificat ainsi que de la clef privée associée pour chaque domaine. En cas de compromission d'un domaine, ses clefs pourraient être exploitées par l'attaquant pour compromettre la confidentialité des échanges. Le contrôle d'accès des domaines doit donc veiller à protéger ces ressources critiques.

La confidentialité des entrées consiste à vérifier la propriété 2 pour l'état (η_s, η_u), c'est-à-dire à faire connaître à l'utilisateur les domaines utilisateur qui reçoivent les entrées système. Nous nous appuyons sur la propriété d'attribution des sorties pour faire connaître à l'utilisateur, via une zone de confiance, les domaines qui sont aptes à recevoir ses entrées, ce qui inclut principalement le domaine utilisateur focalisé. Les autres domaines qui peuvent recevoir des entrées doivent également être connus de l'utilisateur, par exemple par convention concernant les entrées correspondant aux raccourcis clavier du gestionnaire de fenêtres.

La confidentialité des sorties consiste à vérifier la propriété 4 pour l'état (η_s, η'_u). L'utilisateur doit connaître les domaines utilisateur qui ont accès aux sorties système. Cette information doit également être transmise par une zone de confiance.

6.3.3 Accomplissement des entrées

L'accomplissement des entrées consiste à vérifier la propriété 1 pour l'état (η_s, η_u). Cette propriété peut être mise à mal si les entrées ne sont pas accessibles aux domaines utilisateur identifiés comme devant les recevoir.

Dans le cas où une entrée système est modifiée, mais que son abstraction reste identique pour les domaines récepteurs, alors l'accomplissement reste vérifié. Cela peut par exemple correspondre à une rupture protocolaire entre deux agents, ce qui implique un changement des entrées système, mais une abstraction correspondant aux attentes de l'utilisateur.

L'utilisateur doit être informé de la bonne réception des entrées utilisateur par les domaines qui les reçoivent. Ceci peut par exemple être reflété à travers la sortie du domaine cible, comme c'est le cas d'un éditeur de texte qui affiche le texte entré. Dans le cas contraire, une zone de confiance doit afficher cette information.

6.3.4 Contenu de la zone de confiance

En pratique, nous pouvons limiter les informations disponibles dans une zone de confiance. Si l'utilisateur connaît les domaines qui reçoivent les entrées utilisateur déterminées, la zone de confiance peut n'afficher que l'association entre les domaines et les entrées utilisateur indéterminées, autrement dit la hiérarchie de domaines contenant les agents focalisés. Nous

pouvons également éluder les informations concernant les domaines de confiance qui ne sont pas les destinations finales des entrées utilisateur. De manière similaire, il n'est pas nécessaire d'informer qu'une sortie utilisateur transite par un domaine de confiance.

6.3.5 Domaines intermédiaires et communications indirectes

Une sortie utilisateur peut provenir d'un premier domaine et être relayée par un domaine intermédiaire. De même, une entrée utilisateur peut être transférée via un domaine intermédiaire. En cas de compromission d'un domaine intermédiaire, ces communications indirectes peuvent remettre en cause la vérification des propriétés d'attribution, d'accomplissement et de confidentialité.

Sans mesure architecturale permettant de discriminer les domaines par lesquels transitent les entrées ou les sorties indirectes, alors il n'est pas possible de vérifier les propriétés d'attribution, d'accomplissement ou de confidentialité de ces entrées et sorties. Dans ce cas, la compromission d'un domaine intermédiaire remet en cause l'exécution sécurisée telle que nous l'avons définie, car il n'est plus possible de faire la différence entre le domaine compromis et ses fils.

Pour être capable de conserver les propriétés de sécurité indépendamment du caractère compromis d'un domaine intermédiaire, il faut que le protocole de communication entre domaines fournisse une capacité de transfert confidentiel et intègre d'entrée et de sortie.

Le protocole Wayland met à disposition l'interface *sub-surface* qui permet à un agent parent de composer sa sortie avec celle d'un agent fils sans qu'il y ait de divulgation des données affichées. En effet, les métadonnées communiquées par l'agent fils ne contiennent pas directement les données à afficher, mais une référence à une zone mémoire du GPU qui contient ces données. L'utilisation de la mémoire du GPU n'est pas une réponse à notre problématique mais le principe de référence pourrait être réutilisé avec un stockage sécurisé des données. Un contrôle d'accès adapté ou l'utilisation de signature et de chiffrement permettrait de transmettre des sorties système tout en assurant leur attribution et leur confidentialité. Une même approche pourrait être envisagée pour assurer l'accomplissement et l'intégrité des entrées de manière indirecte.

6.3.6 Prise en compte de la latence d'interaction

La latence des transmissions des entrées utilisateur et de celle de l'interprétation des sorties utilisateur peut poser des problèmes dus à des prises en compte des entrées utilisateur différentes de ce que souhaite l'utilisateur. Ces *race-conditions* peuvent en effet poser des problèmes de type TOCTOU pour l'utilisateur.

Par exemple, lorsque l'utilisateur souhaite cliquer sur un bouton pour faire une action particulière sur une application, une autre fenêtre d'application peut surgir subrepticement et recevoir cette entrée utilisateur, ce qui n'était pas l'intention initiale de l'utilisateur. Dans notre modèle, le problème se pose si un agent d'un premier domaine utilisateur perd une entrée utilisateur au profit d'un agent d'un autre domaine utilisateur.

Une implémentation instanciant notre modèle doit intégrer une dimension temporelle dans l'abstraction d'une sortie système. Nous imposons l'hypothèse que l'implémentation d'une sortie utilisateur inclut le temps de réception de l'information ainsi que le temps d'interprétation de l'utilisateur. De même, nous imposons l'hypothèse que l'implémentation d'une entrée utilisateur inclut le temps de réaction de l'utilisateur ainsi que le temps nécessaire pour générer cette entrée. Le respect de ces contraintes implique que l'utilisateur ait suffisamment de temps pour interpréter correctement tout changement critique lié aux domaines.

En pratique, les temps utilisés peuvent être soit estimés, soit implicitement déduits par des contraintes sur l'utilisation de l'IHM. Par exemple, si le changement de domaine utilisateur n'est possible que par une action de l'utilisateur, elle-même conditionnée par une sortie de confiance, alors l'utilisateur effectue cette action de changement de domaine en connaissance de cause. L'utilisateur est ainsi directement responsable de l'actualisation de sa compréhension des domaines dans le temps.

6.4 Mise en pratique

6.4.1 Évaluation de l'IHM de Clip OS

Nous avons détaillé dans la section 6.2.2 l'instanciation de notre modèle sur Clip OS, notamment concernant l'abstraction des agents et de leurs domaines. Les domaines utilisateur sont : *Socle*, *Bas* et *Haut*.

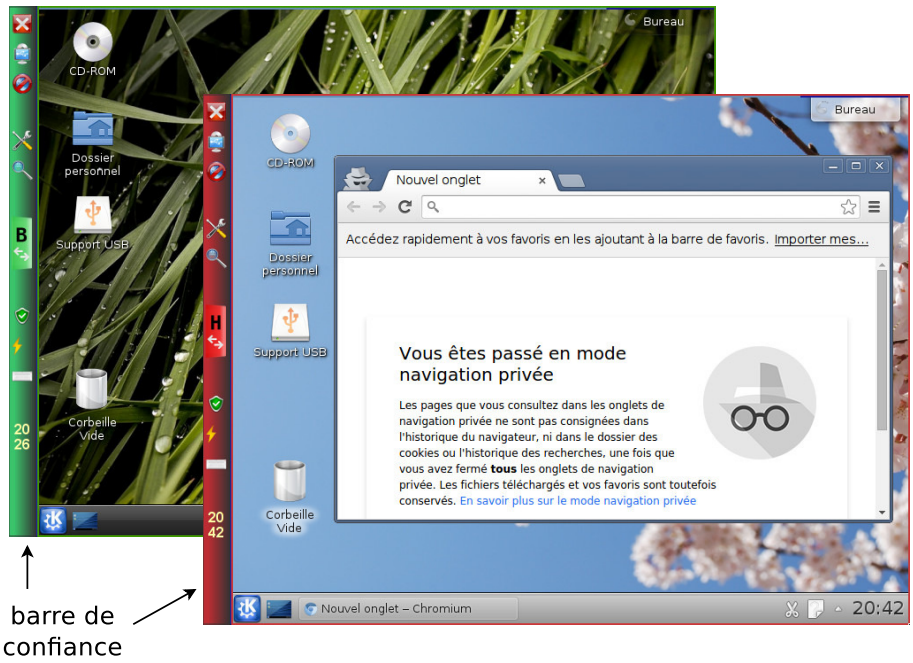


Figure 6.5 – Captures d'écran de Clip OS lors de l'utilisation du domaine *Bas* ou *Haut*

La figure 6.5 montre l'affichage d'un même écran physique, soit lors de l'utilisation du domaine *Bas* (à gauche), soit lors de l'utilisation du domaine *Haut* (à droite).

Affichage et attribution des sorties

Les agents des trois domaines *Bas*, *Haut* et *Socle* sont autorisés à afficher leurs sorties système. Le domaine de confiance *Socle* peut afficher ses sorties sur tout l'affichage. Ce domaine contient des visionneuses VNC, c'est-à-dire des agents dédiés à l'affichage des sorties des domaines *Bas* et *Haut*. Ces visionneuses permettent notamment de créer des régions d'affichage autorisées pour ces deux domaines utilisateur.

Comme détaillé dans la section 5.4.2, le système Clip OS apporte la ségrégation en domaine graphique via une modification du serveur *X.Org*. Sur cette base, nous avons implémenté une barre de confiance qui sert trois objectifs : représenter une zone de confiance, basculer entre le domaine *Bas* et *Haut*, et permettre le lancement d'applications de configuration dans le domaine *Socle*. Cette barre de confiance fait partie du domaine de confiance *Socle*.

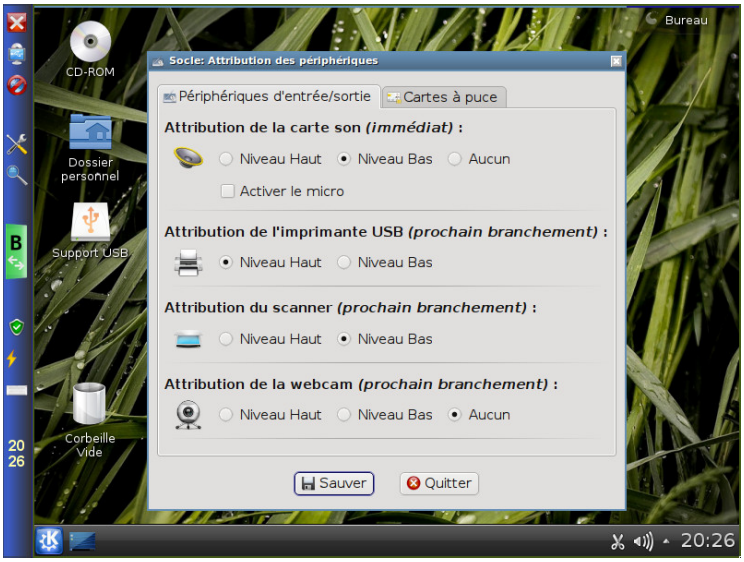


FIGURE 6.6 – Captures d'écran de CLIP OS lors de la configuration du domaine *Socle*

La barre de confiance est visible à gauche de chaque capture d'écran, emplacement statique et connu au préalable par l'utilisateur. La barre de confiance reflète le domaine utilisateur focalisé, c'est-à-dire celui qui contient des agents capables d'exhiber leurs sorties sur la partie droite de l'écran, et qui reçoivent les entrées utilisateur indéterminées. Cette identification est possible par la couleur de la barre de confiance qui correspond à celle du domaine ainsi que par la lettre affichée : verte et *B* pour *Bas* ou rouge et *H* pour *Haut*. En complément, chaque domaine utilisateur est entouré d'une bordure colorée correspondant à la couleur du domaine.

Sur la figure 6.3, l'agent a_s^1 est celui qui gère cette barre de confiance. Les domaines *Bas* et *Haut* n'ont pas accès aux sorties du domaine *Socle*, qui incluent celles de la barre de confiance.

Un cas qui n'est pas pris en charge par la barre de confiance de CLIP OS est l'affichage des autres agents du domaine *Socle* par dessus les domaines *Bas* ou *Haut*. Étant donné que la barre de confiance reflète uniquement le domaine focalisé, elle n'indique pas explicitement le domaine qui affiche sur la partie droite de l'écran. Sur la capture d'écran de la figure 6.6, l'application de configuration des périphériques est focalisée ce qui est bien indiqué par la barre de confiance, mais une partie du niveau *Bas* est toujours visible à l'arrière-plan. Ce comportement peut être davantage problématique lorsqu'une application du domaine *Socle* est ouverte et explicitement défocalisée par l'utilisateur, via un clic sur l'affichage du domaine en arrière-plan, ce qui a pour conséquence d'avoir une barre de confiance reflétant le domaine *Bas*. Cette approche a été privilégiée pour deux raisons. La première est de simplifier l'interface utilisateur en n'affichant pas explicitement la liste des applications lancées par le domaine *Socle* tout en permettant à l'utilisateur de reprendre la main sur ces applications. La deuxième raison est de permettre à l'utilisateur de visualiser simultanément une application du domaine *Socle* et une application du domaine *Bas* ou *Haut*, ce qui permet par exemple de visualiser une documentation de configuration à partir d'un domaine de travail en même temps que de modifier la configuration du domaine *Socle*. Par ailleurs, l'attribution des sorties du domaine *Socle* est facilitée par le fait qu'elles ne sont pas inscriptibles par les autres domaines. Cette approche est un compromis pour simplifier l'information mise à disposition de l'utilisateur tout en limitant fortement le risque de mauvaise identification du domaine visible sur la partie droite de l'écran.

Originalité et connaissance des entrées

Le domaine *Socle* ne peut recevoir des entrées système que par les HID connectés au système. Un seul *seat* est possible et tout ajout ou suppression de HID est signalé à l'utilisateur par une notification. Cette notification n'est cependant que partiellement affichée par une zone de confiance. L'utilisateur a également la possibilité de choisir la configuration de ses claviers et souris. Il a donc connaissance de ses entrées.

Confidentialité des entrées et des sorties

La barre de confiance reflète le domaine de travail en cours d'utilisation. Ce domaine est chargé de transmettre les entrées indéterminées au domaine focalisé. Les raccourcis clavier pris en compte par le domaine *Socle*, qui incluent le changement de domaine et le verrouillage de session, sont considérés comme connus de l'utilisateur.

L'utilisation des visionneuses implique une rupture protocolaire entre les domaines de travail et le domaine *Socle*. Les seuls messages reconnus parmi ceux envoyés par un client VNC sont des sorties système. Les visionneuses, et donc le domaine *Socle*, ont accès à ces sorties. Par contre, un autre domaine de travail, potentiellement compromis, n'a accès qu'à ses propres sorties. L'utilisateur peut ne pas avoir connaissance que le domaine *Socle* a accès aux sorties des domaines *Bas* ou *Haut*. Pour autant, le domaine *Socle* est un domaine de confiance qui fait partie de la TCB. Comme expliqué dans la section 6.3.4, il est acceptable de l'exclure des domaines dont l'utilisateur doit avoir connaissance.

Accomplissement des entrées

De la même manière que pour la focalisation, les entrées indéterminées sont transférées au domaine focalisé : *Socle*, *Bas* ou *Haut*. L'utilisateur en est informé via la barre de confiance.

Domaines intermédiaires et communications indirectes

Concernant les domaines potentiellement non de confiance, le système Clip OS n'utilise pas de domaine intermédiaire.

Prise en compte de la latence d'interaction

Concernant des attaques de type TOCTOU, la solution implémentée dans Clip OS est de n'effectuer des changements de domaines utilisateur que lors d'une action utilisateur. Cette action peut être un appui sur le bouton *H/B* situé au centre de la barre de confiance, un raccourci clavier (SAK) ou le lancement d'une application du domaine *Socle*. Dans tous les cas, un changement de domaine est initié par l'utilisateur, ce qui implique qu'il ait connaissance de ce changement de domaine lors de l'émission d'une entrée utilisateur correspondante.

Exécution sécurisée

Comme décrit dans la section 6.2.3 (page 102), une exécution sécurisée prend en compte l'accomplissement et la confidentialité d'entrée (propriétés 1 et 2) ainsi que l'attribution et la confidentialité des sorties (propriétés 3 et 4) lors des transitions $(\eta_s, \eta_u) \xrightarrow{\overline{i_s}} (\eta'_s, \eta'_u)$ et $(\eta_s, \eta_u) \xrightarrow{\overline{o_s}} (\eta_s, \eta'_u)$. Nous venons de voir ce que Clip OS représente à l'utilisateur afin de synchroniser les états utilisateur avec les états système tout au long d'une session utilisateur.

Comme indiqué sur la figure 6.3 (page 98), l'abstraction des domaines système correspond à *Socle*, *Bas* ou *Haut*.

Dans le cas de l'accomplissement des entrées (propriété 1), le résultat de $\overline{\text{RECEIVER}(\eta_s, i_s)}$ correspond à l'ensemble des domaines utilisateur qui reçoivent les entrées utilisateur indéterminées. Dans le cas où un agent du domaine *Socle* (autre qu'une visionneuse) est focalisé, alors cet ensemble contient *Socle*. Dans le cas où le domaine *Bas* ou *Haut* est focalisé, alors cet ensemble contient le domaine focalisé ainsi que *Socle*. Cependant, étant donné que *Socle* est un domaine de confiance, même si l'utilisateur n'a pas conscience de son implication, la présence de ce domaine dans l'ensemble n'est pas problématique. Étant donné que les informations exposées à l'utilisateur correspondent à celles du système, concernant la propriété 1 mais également la propriété 2, nous notons : $\{Socle\} \cup \overline{\text{RECEIVER}(\eta_u, \overline{i_s})} = \{Socle\} \cup \overline{\text{RECEIVER}(\eta_s, i_s)}$.

La propriété 3 est applicable à l'état (η_s, η'_u) avec η'_u correspondant à ce que l'utilisateur se représente après avoir vu une sortie. L'utilisateur a connaissance des domaines qui affichent sur l'écran grâce à la barre de confiance. Celle-ci contient les informations correspondantes au retour de $\overline{\text{DISPLAYER}(\eta_s, o_s)}$, hormis le cas présenté avec la figure 6.6, ce qui nous donne $\{Socle\} \cup \overline{\text{DISPLAYER}(\eta'_u, \overline{o_s})} = \{Socle\} \cup \overline{\text{DISPLAYER}(\eta_s, o_s)}$.

La propriété 4 concerne l'observation de sorties et s'applique sur les mêmes états que la propriété 3. Un poste CLIP OS (en production) n'offre pas de fonctionnalité de captures d'écran globales, c'est-à-dire qui peuvent inclure des sorties de *Socle* ou de plus d'un domaine utilisateur. Cependant le domaine *Socle* a besoin d'observer les sorties des domaines *Bas* et *Haut* pour les retranscrire sur l'écran. Nous considérons que l'utilisateur sait qu'un domaine peut observer ses propres sorties, ce qui nous donne : $\{Socle\} \cup \overline{\text{WATCHER}(\eta_s, o_s)} = \{Socle\} \cup \overline{\text{WATCHER}(\eta'_u, \overline{o_s})}$.

6.4.2 Attaque sur une implémentation d'attribution sans zone de confiance

Comme nous l'avons vu, l'attribution d'une sortie utilisateur nécessite l'identification par l'utilisateur des domaines qu'il manipule. Contrairement à CLIP OS, certains systèmes n'utilisent pas de zone de confiance pour afficher l'attribution.

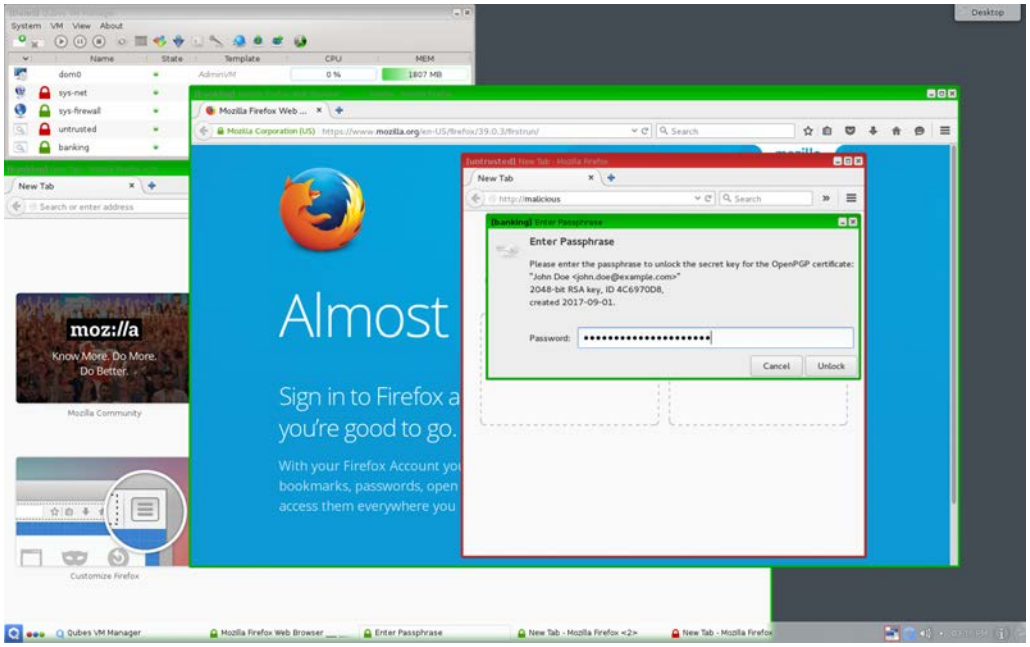


FIGURE 6.7 – Capture d'écran lors de l'utilisation de plusieurs domaines avec Qubes OS

La figure 6.7 montre l'interface graphique de Qubes OS [38]. Chaque fenêtre et menu contextuel dessiné par un agent est entouré d'une bordure colorée correspondant à son domaine utilisateur. Chacun de ces domaines utilisateur est un fils du domaine de l'agent racine responsable de l'affichage du bureau. Comme l'ont introduit TX [70, 71] puis Trusted Solaris [82], le titre de chaque fenêtre contient également le nom du domaine à l'origine du contenu de cette fenêtre. Enfin, les fenêtres qui ne reçoivent pas les entrées utilisateur indéterminées peuvent être assombries. Ces différentes informations correspondent bien à celles qui doivent être remontées à l'utilisateur, typiquement via une zone de confiance.

Cependant, pour une IHM de confiance, ces informations doivent être transmises via des sorties utilisateur non inscriptibles par des domaines potentiellement corruptibles. Ces conditions sont respectées si la sortie utilisateur de la bordure et du titre n'est jamais inscriptible par un domaine utilisateur autre qu'un domaine de confiance, ce qui n'est pas le cas sur la figure 6.7. En effet, étant donné que les fenêtres peuvent être déplacées, redimensionnées et superposées, la propriété d'attribution des sorties n'est pas vérifiée, ce qui met à mal l'identification de domaine par une attaque de catégorie *picture-in-picture*. Dans cet exemple, l'utilisateur peut facilement être trompé si le navigateur du domaine rouge dessine une fenêtre avec une bordure verte demandant un mot de passe. Si l'utilisateur ne fait pas attention qu'il n'y a pas d'assombrissement de la barre de titre de la fenêtre du navigateur, alors l'état utilisateur η_u n'est plus synchronisé avec l'état système η_s, ce qui peut ici avoir un impact sur la propriété d'accomplissement et donc de confidentialité des entrées lorsque l'utilisateur va entrer son mot de passe.

6.5 Conclusion

La démarche présentée dans ce chapitre a permis de préciser les éléments permettant de mettre en place une politique de sécurité sur l'interface homme-machine. Nous avons formalisé le point de vue de l'utilisateur concernant le système via des abstractions. Ces abstractions nous ont ensuite permis de définir les propriétés de sécurité qu'une IHM de confiance doit vérifier pour être capable d'interagir de manière sécurisée avec l'utilisateur, ainsi que la confidentialité et l'intégrité des informations échangées. Au sein d'une IHM qui vérifie ces propriétés et d'un utilisateur qui prend en compte les informations qui lui sont fournies, des attaques de type *confused deputy* sur l'utilisateur seront mises en échec.

Nous avons ensuite expliqué des moyens d'implémentation de ces propriétés de sécurité, notamment avec une zone de confiance, élément indispensable pour permettre à l'utilisateur d'attribuer l'origine de la source d'éléments graphiques. Ceci nous a permis d'évaluer l'IHM de Clip OS avec notre modèle. Enfin, nous avons expliqué un exemple d'attaque lorsque notre modèle n'est pas correctement instancié.

Pour qu'un système qui interagit avec des utilisateurs soit réellement sécurisé, il doit vérifier la propriété d'exécution sécurisée de notre modèle. Afin de compléter sa politique de sécurité, un tel système pourrait également intégrer la notion de domaines utilisateur ainsi que les différents composants de l'IHM tels que nous les avons définis.

Conclusion

Problématique et solutions apportées

Nous nous sommes intéressés à la protection de l'utilisateur et plus particulièrement de ses données vis-à-vis d'une application malveillante. L'objectif était de limiter la fuite et la modification des données utilisateur. Pour cela, notre travail a consisté à définir un contrôle d'accès qui soit adapté à l'utilisateur, c'est-à-dire un contrôle d'accès invisible lors du respect de la politique de sécurité. Nous présentons un modèle d'IHM sécurisée pour d'une part étendre le contrôle d'accès à l'utilisateur et d'autre part le protéger contre des attaques via cette interface.

L'application du contrôle d'accès est aujourd'hui principalement adaptée aux besoins de l'administrateur d'un système pour sa protection et la protection des utilisateurs entre eux. Sur un système GNU/Linux, un utilisateur non-administrateur ne dispose pas de l'opportunité de configurer un contrôle d'accès qui s'applique à ses applications. L'interaction de l'utilisateur avec le système via l'IHM est rarement prise en compte et aucun modèle de sécurité n'existait pour valider la sécurité d'une IHM.

Le prérequis de nos travaux est l'utilisation d'une TCB qui comprend le noyau du système d'exploitation ainsi que les services nécessaires au bon fonctionnement du système. Le contrôle d'accès appliqué par le système doit donc rester fonctionnel. Enfin, l'environnement physique du système est également considéré comme de confiance.

La première partie de cette thèse concerne le contrôle d'accès adapté à l'utilisateur. Nous avons créé un modèle permettant, à partir d'une configuration initiale, d'identifier les activités de l'utilisateur en fonction de son utilisation du système. Les activités utilisateur sont ensuite utilisées pour transiter entre des domaines de sécurité. Une transition d'un domaine à l'autre est autorisée dans le seul cas où l'une des activités potentielles du domaine source autorise les nouveaux accès obtenus par le domaine destination. Cette approche permet une mise en place non intrusive du contrôle d'accès pour l'utilisateur.

StemJail implémente ce contrôle d'accès adaptatif sur un système GNU/Linux. Les résultats de cette preuve de concept ont permis de valider notre modèle et son application sur un système réel actuel. En effet, le développement de *StemJail* a permis de valider l'utilisation des espaces de noms Linux pour faire du cloisonnement dynamique non privilégié avec un faible impact sur les performances. Cette implémentation a également permis d'identifier des limites à l'utilisation des espaces de noms Linux, ce qui a mené nos recherches à l'extension des fonctionnalités de sécurité du noyau Linux avec *Landlock*.

Le nouveau contrôle d'accès apporté par le développement du module de sécurité *Landlock* permet de limiter les accès d'un processus à un sous-ensemble des accès autorisés par son parent. La configuration des règles de sécurité est décrite sous forme de programmes eBPF permettant une très grande flexibilité tout en limitant la surface d'attaque du noyau. Suite à l'intérêt suscité par les communautés de la sécurité et des conteneurs, *Landlock* est en cours d'intégration dans la version de référence de Linux.

La deuxième partie de cette thèse s'intéresse à la problématique d'IHM sécurisée. Une

interface de confiance est primordiale dans l'interaction entre le système et l'utilisateur afin que l'utilisateur identifie de manière fiable les domaines avec lesquels il interagit. Nous avons défini trois propriétés de sécurité concernant les entrées utilisateur : l'originalité, l'accomplissement et la confidentialité. Nous avons également défini trois propriétés de sécurité concernant les sorties utilisateur : l'affichage, l'attribution et la confidentialité. Nous avons formalisé une exécution sécurisée d'IHM avec un graphe comportant des états système et utilisateur pour lesquels les propriétés de sécurité sur les entrées et sorties doivent être vérifiées. La vérification de ces propriétés permet de se protéger contre des attaques portant sur l'interface et les fonctionnalités fournies par l'IHM, mais également sur l'utilisateur, comme cela peut être le cas avec une attaque de type *confused deputy*. Notre modèle est illustré avec le système Clip OS, ce qui nous permet d'évaluer la sécurité de son IHM.

Discussion et travaux futurs

Notre approche du contrôle d'accès avec *StemJail*, *Landlock* et la définition d'IHM de confiance apporte des réponses à la problématique du confinement d'applications par activité. Les navigateurs web sont également une cible particulièrement pertinente pour la différenciation des activités utilisateur et leur retranscription graphique. Ils gèrent différentes fenêtres et onglets qui correspondent à autant de domaines utilisateur. Le *sandboxing* dont ils peuvent profiter pourrait également être amélioré avec les fonctionnalités offertes par *Landlock*.

Deux autres problématiques complémentaires liées au contrôle d'accès sont envisagées : la représentation globale des politiques appliquées sur un système et l'audit de cette composition de règles.

Avoir une vue globale de la composition des différentes politiques de sécurité appliquées sur le système permet de valider la cohérence globale et d'effectuer des opérations d'optimisation comme la suppression de règles redondantes. Que les politiques de sécurité proviennent de l'administrateur ou des développeurs, la sémantique de la composition de ces règles permet de définir une politique globale de sécurité pour un système entièrement maîtrisé. Par exemple, la définition d'une politique globale pertinente pourrait prendre en compte la sémantique des contrôles d'accès mis en place pour le système, avec celle des contrôles d'accès mis en place par un navigateur web. Il pourrait alors y avoir un prolongement des règles de *Same-Origin Policy* [83] (SOP) avec celles de la politique d'accès aux processus, aux fichiers ou au réseau.

De manière complémentaire, l'aptitude à auditer une politique de sécurité est importante. Il est difficile de vérifier la cohérence des contraintes mises en place avec des programmes *Landlock*. La définition d'un objet et son identification est programmatique et potentiellement dynamique. Par exemple, il n'est pas simple d'identifier tous les fichiers qui seront acceptés par une règle qui effectue une vérification de l'arborescence. Ces propriétés sont interprétées lors d'un accès demandé par un processus cloisonné, ce qui complique l'extraction de ces conditions pour les évaluer indépendamment du contexte du processus.

Une approche prometteuse pour ces problématiques consiste à considérer les règles *Landlock* comme un mécanisme générique de moteur de contrôle d'accès généré par un langage dédié (*Domain Specific Language* : DSL) de plus haut niveau. Ce DSL pourrait être analogue à des langages conçus pour faire du contrôle d'accès, comme celui utilisé par SELinux. Cette approche permettrait la vérification de différentes contraintes de sécurité sur une politique. Une utilisation plus avancée serait de contraindre l'utilisation de ce DSL par les sujets via un service système dédié pour forcer le respect de propriétés de sécurité. L'architecture de *Landlock* permet une telle approche grâce au découpage de la création de règles avec l'appel système `bpf`, de celui de l'application d'une règle via l'appel système

`seccomp`.

De manière analogue, l'intégration d'une politique de sécurité d'IHM dans la politique globale du système serait une suite logique. Pour cela, l'utilisation d'un DSL adapté pour définir les objets d'IHM comme des zones d'affichage ou des ensembles d'entrées utilisateur est nécessaire. Comme dans le cas de *Landlock*, l'intégration de ces règles dans une vue globale permettrait d'exécuter des requêtes d'audit sur une composition de politiques d'accès aux ressources du système et à celles de l'IHM.

Bibliographie

[1] Mickaël Salaün. StemJail : Cloisonnement dynamique d'activités pour la protection des données utilisateur. In *Symposium sur la sécurité des technologies de l'information et des communications*, 2015.

[2] Mickaël Salaün, Marion Daubignard, and Hervé Debar. StemJail : Dynamic Role Compartmentalization. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016.

[3] Rust project developers. The Rust Programming Language, 2010.

[4] Mickaël Salaün. Landlock LSM : Unprivileged sandboxing. In *Kernel Recipes*, 2016.

[5] Mickaël Salaün. Landlock : cloisonnement programmable non privilégié. In *Symposium sur la sécurité des technologies de l'information et des communications*, 2017.

[6] Mickaël Salaün. Landlock LSM : toward unprivileged sandboxing. In *Linux Security Summit*, 2017.

[7] Mickaël Salaün. File access-control per container with Landlock. In *FOSDEM*, 2018.

[8] Vincent Strubel. CLIP : une approche pragmatique pour la conception d'un OS sécurisé. In *Symposium sur la sécurité des technologies de l'information et des communications*, 2015.

[9] Department of Defense (USA). Trusted Computer System Evaluation Criteria (Orange Book). 1985.

[10] D. Elliott Bell and Leonard J. LaPadula. Secure Computer Systems : Mathematical Foundations. Technical report, MITRE Corp., 1973.

[11] Kenneth J. Biba. Integrity Considerations for Secure Computer Systems. Technical report, DTIC Document, 1977.

[12] David F. C. Brewer and Michael J. Nash. The Chinese Wall Security Policy. In *Symposium on Security and Privacy*. IEEE, 1989.

[13] David Ferraiolo and Richard Kuhn. Role-Based Access Control. In *NIST-NCSC*, 1992.

[14] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. A Domain and Type Enforcement UNIX Prototype. In *Proceedings of the 5th USENIX UNIX Security Symposium*, 1995.

[15] 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. In *Proceedings of the 6th USENIX UNIX Security Symposium*, 1996.

[16] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS : a fast capability system. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*. ACM, 1999.

[17] Z. C. Schreuders, T. McGill, and C. Payne. The State of the Art of Application Restrictions and Sandboxes : A Survey of Application-oriented Access Controls and their Shortfalls. *Computers & Security*, 2013.

[18] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *USENIX Annual Technical Conference, FREENIX Track*, 2001.

[19] Dionysus Blazakis. The Apple Sandbox. In *Black Hat DC*, 2011.

[20] Robert Watson, Wayne Morrison, Chris Vance, and Brian Feldman. The TrustedBSD MAC Framework : Extensible Kernel Access Control for FreeBSD 5.0. In *USENIX Annual Technical Conference, FREENIX Track*, 2003.

[21] Will Drewry. Dynamic seccomp policies (using BPF filters), 2012. `https://lwn.net/Articles/475019/`.

[22] Will Drewry. Seccomp BPF (SECure COMPuting with filters), 2012. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/prctl/seccomp_filter.txt?h=v3.5`.

[23] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum : practical capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[24] The FreeBSD Project. Casper, 2013.

[25] Theo de Raadt. Pledge : a new mitigation mechanism. In *Hackfest*, 2015.

[26] Taesoo Kim and Nickolai Zeldovich. Practical and Effective Sandboxing for Non-root Users. In *USENIX Annual Technical Conference*, 2013.

[27] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker). In *Proceedings of the 6th USENIX UNIX Security Symposium*, 1996.

[28] Niels Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[29] Zhenkai Liang, V. N. Venkatakrishnan, and R. Sekar. Isolated Program Execution : An Application Transparent Approach for Executing Untrusted Programs. In *Proceedings of 19th Annual Computer Security Application Conference*. IEEE, 2003.

[30] Tal Garfinkel. Traps and Pitfalls : Practical Problems in System Call Interposition Based Security Tools. In *NDSS*, 2003.

[31] Mark Russinovich. Inside Windows Vista User Account Control. *Microsoft TechNet Magazine*, 2007.

[32] Steven Furnell and Kerry-Lynn Thomson. Recognising and addressing 'security fatigue'. *Computer Fraud & Security*, 2009.

[33] Mickaël Salaün. StemJail, 2015. `https://github.com/stemjail`.

[34] Sukadev Bhattiprolu, Eric W. Biederman, Serge Hallyn, and Daniel Lezcano. Virtual Servers and Checkpoint/Restart in Mainstream Linux. *ACM SIGOPS Operating Systems Review*, 2008.

[35] Norm Hardy. The Confused Deputy (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 1988.

[36] Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. Engineering the Servo Web Browser Engine using Rust. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016.

[37] Norman Feske and Christian Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *Proceedings of the 21st Annual Computer Security Applications Conference*. IEEE, 2005.

[38] Joanna Rutkowska and Rafal Wojtczuk. Qubes OS Architecture. 2010.

[39] Anurag Acharya and Mandar Raje. MAPbox : Using Parameterized Behavior Classes to Confine Untrusted Applications. In *Proceedings of the 9th USENIX Security Symposium*, 2000.

[40] Matt Bishop and Michael Dilger. Checking for Race Conditions in File Accesses. *Computing Systems*, 1996.

[41] Shaya Potter and Jason Nieh. Apiary : Easy-to-Use Desktop Application Fault Containment on Commodity Operating Systems. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[42] Piero Bonatti, Sabrina De Capitani di Vimercati, and Pierangela Samarati. A Modular Approach to Composing Access Control Policies. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*. ACM, 2000.

[43] Piero Bonatti, Sabrina de Capitani di Vimercati, and Pierangela Samarati. An Algebra for Composing Access Control policies. *ACM Transactions on Information and System Security*, 2002.

[44] Steven McCanne and Van Jacobson. The BSD Packet Filter : A New Architecture for User-level Packet Capture. In *USENIX Annual Technical Conference, FREENIX Track*, 1992.

[45] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux Security Modules : General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

[46] J. Vollbrecht, P. Calhoun, S. Farrell, L. Gommans, G. Gross, B. de Bruijn, C. de Laat, M. Holdrege, and D. Spence. RFC 2904 : AAA Authorization Framework. *Request For Comment, Network Working Group*, 2000.

[47] Elena Reshetova, Filippo Bonazzi, and N. Asokan. Randomization can't stop BPF JIT spray. In *Black Hat Europe*, 2016.

[48] Daniel Borkmann. bpf : add generic constant blinding for use in jits. In *Linux commit 4f3446bb809f*, 2016.

[49] Alexei Starovoitov. What do we call an explosive mix of LSM, cgroup, BPF, seccomp ? In *Linux Plumbers Conference*, 2016.

[50] P. T. Cummings, D. A. Fullan, M. J. Goldstien, M. J. Gosse, J. Picciotto, J. P. L. Woodward, and J. Wynn. Compartmented Model Workstation : Results Through Prototyping. In *Symposium on Security and Privacy*. IEEE, 1987.

[51] Jeffrey L. Berger, Jeffrey Picciotto, John P. L. Woodward, and Paul T. Cummings. Compartmented Mode Workstation : Prototype Highlights. *IEEE Transactions on Software Engineering*, 1990.

[52] X Consortium. X.org xorg-docs documentation, 2013.

[53] Kristian Høgsberg. Wayland Protocol Specification, 2008.

[54] Ka-Ping Yee. User Interaction Design for Secure Systems. In *Information and Communications Security, 4th International Conference, ICICS*. Springer, 2002.

[55] Bernhard Beckert and Gerd Beuster. A Method for Formalizing, Analyzing, and Verifying Secure User Interfaces. In *International Conference on Formal Engineering Methods*, 2006.

[56] Gregory Conti, Mustaque Ahamad, and John Stasko. Attacking Information Visualization System Usability Overloading and Deceiving the Human. In *Proceedings of the 2005 Symposium on Usable Privacy and Security*. ACM, 2005.

[57] Lin-Shung Huang, Alexander Moshchuk, Helen J. Wang, Stuart Schecter, and Collin Jackson. Clickjacking : Attacks and Defenses. In *Proceedings of the 21th USENIX Security Symposium*, 2012.

[58] Thomas Fischer, Ahmad-Reza Sadeghi, and Marcel Winandy. A Pattern for Secure Graphical User Interface Systems. In *20th International Workshop on Database and Expert Systems Application*. IEEE, 2009.

[59] Collin Jackson, Daniel R. Simon, Desney S. Tan, and Adam Barth. An Evaluation of Extended Validation and Picture-in-Picture Phishing Attacks. In *International Conference on Financial Cryptography and Data Security*, 2007.

[60] Franziska Roesner. *Security and Privacy for Untrusted Applications in Modern and Emerging Client Platforms*. PhD thesis, 2014.

[61] Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. Peeking into Your App without Actually Seeing It : UI State Inference and Novel Android Attacks. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[62] Marc Stiegler. An Introduction to Petname Systems. In *Advances in Financial Cryptography*, 2005.

[63] David Wagner and Dean Tribble. A Security Analysis of the Combex DarpaBrowser Architecure. Technical report, 2002.

[64] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan. User-Driven Access Control : Rethinking Permission Granting in Modern Operating Systems. In *Symposium on Security and Privacy*. IEEE, 2012.

[65] Eamon Walsh. Application of the Flask Architecture to the X Window System Server. In *Security Enhanced Linux Symposium*, 2007.

[66] Tiago Vignatti. libxcb and X11R7.7 client API, 2012.

[67] Microsoft. Windows MSDN - Desktop.

[68] Google. Chromium Windows Sandbox.

[69] Crispin Cowan. Windows 8 Security : Supporting User Confidence, 2013.

[70] Jeremy Epstein. A High-Performance Hardware-Based High Assurance Trusted Windowing System. In *Proceedings of the 19th National Information Systems Security Conference*, 1996.

[71] Jeremy Epstein. Fifteen Years After TX : A Look Back at High Assurance Multi-Level Secure Windowing. In *Proceedings of the 22nd Annual Computer Security Applications Conference*. IEEE, 2006.

[72] Norman Feske and Hermann Härtig. DOpE - a Window Server for Real-Time and Embedded Systems. Technical report, 2003.

[73] Norman Feske and Hermann Härtig. DOpE - a Window Server for Real-Time and Embedded Systems. In *Real-Time Systems Symposium*. IEEE, 2003.

[74] Norman Feske. *Securing Graphical User Interfaces*. PhD thesis, Technische Universität Dresden, 2009.

[75] Jonathan S. Shapiro, John Vanderburgh, Eric Northup, and David Chizmadia. Design of the EROS Trusted Window System. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[76] Agence nationale de la sécurité des systèmes d'information. CLIP OS, 2006.

[77] Genode, 2011. `https://genode.org/`.

[78] Hamed Okhravi and David M. Nicol. TrustGraph : Trusted Graphics Subsystem for High Assurance Systems. In *Proceedings of the 25th Annual Computer Security Applications Conference*. IEEE, 2009.

[79] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramon Caceres, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert Van Doorn. Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor. In *Proceedings of the 21st Annual Computer Security Applications Conference*. IEEE, 2005.

[80] Mark Anderson, Chris North, John Griffin, Robert Milner, John Yesberg, and Kenneth Yiu. Starlight : Interactive Link. In *Proceedings of 12th Annual Computer Security Application Conference*. IEEE, 1996.

[81] Mark Beaumont, Jim McCarthy, and Toby Murray. The Cross Domain Desktop Compositor : Using hardware-based video compositing for a multi-level secure user interface. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016.

[82] Glenn Faden. Solaris Trusted Extensions. Technical report, Sun Microsystems, 2006.

[83] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. Protecting Browser State from Web Privacy Attacks. In *Proceedings of the 15th International Conference on World Wide Web*. ACM, 2006.