Défense contre les attaques à l'aune des nouvelles formes de virtualisation des infrastructures

Maxime Bélair

DOCTORAT BRETAGNE LOIRE / MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

THÈSE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPÉRIEURE
MINES-TÉLÉCOM ATLANTIQUE BRETAGNE
PAYS-DE-LA-LOIRE - IMT ATLANTIQUE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Maxime BÉLAIR

Défense contre les attaques à l'aune des nouvelles formes de virtualisation des infrastructures

Thèse présentée et soutenue à l'IMT Atlantique, Nantes, le 08/12/2021
Unité de recherche : LS2N
Thèse N° : 2021IMTA0279

Rapporteurs avant soutenance :

Alain TCHANA	Professeur, ENS Lyon
Marie-Laure POTET	Professeure, ENSIMAG

Composition du Jury :

Président :	Aurélien FRANCILLON	Professeur associé, EURECOM
Examinateurs :	Jérémy BRIFFAUT	Maître de conférences, INSA CVL
	Aurélien FRANCILLON	Professeur associé, EURECOM
	Marie-Laure POTET	Professeure, ENSIMAG
	Alain TCHANA	Professeur, ENS Lyon
	Gaël THOMAS	Professeur, Télécom SudParis
Dir. de thèse :	Jean-Marc MENAUD	Professeur, IMT Atlantique
Co-dir. de thèse :	Sylvie LANIEPCE	Ingénieure de recherche, Orange Labs

Remerciements

En entamant cette thèse, j'avais pleinement conscience que le travail qui m'attendait serait difficile et que tout restait à construire. J'ignorais en revanche à quel point je serais entouré de personnes bienveillantes et intéressantes tout au long de cette thèse. Je n'aurais probablement pu réussir ce travail doctoral, particulièrement dans le contexte de la Covid-19 sans leur soutien à toute épreuve.

Ainsi, je tiens à remercier tout particulièrement Sylvie Laniepce pour son encadrement de grande qualité tout au long de ces trois ans. Merci Sylvie d'avoir toujours su me challenger intellectuellement et me pousser à tirer le meilleur de moi-même, toujours dans la bienveillance.

Merci également à Jean-Marc Menaud d'avoir accepté d'encadrer cette thèse et ce malgré la distance géographique imposée par les contraintes d'une thèse CIFRE.

Je souhaiterais remercier mes rapporteurs de thèse pour leur temps et leur commentaires pertinents. Je souhaiterais également remercier mon jury de thèse d'avoir accepté de juger mon travail et pour leurs questions durant ma soutenance.

Je tiens par ailleurs à remercier les doctorants de l'équipe SPI : Adel, Aïda, Anaïs, Donald, Guillaume, Olivier, Paul et Takoua d'avoir partagé un bout de cette aventure humaine avec moi. Merci pour ces discussions interminables, ces fous rires, ces pauses café et ce soutien dans les quelques moments plus difficiles.

Merci à mes amis de Bourges, Blois et d'Angoulême : Adrien(s), Arthur, Bastien, Flavien, Guillaume, Jordan, Killian, Landry, Lorin, Martin, Merwann, Pierre-Henri, Rémi, et Romain, pour les bons moments passés. Ne changez rien, vous êtes les meilleurs.

Merci à Mélanie, de m'avoir accompagné tout au long de cette aventure, malgré les difficultés et les péripéties rencontrées. Merci d'avoir toujours cru en moi même dans les moments difficiles.

Enfin, je tiens à remercier mes parents et mon frère qui ont su m'accompagner avec patience et encouragements. Je tiens à m'excuser sincèrement de les avoir contraints à se familiariser avec la sécurité informatique. J'aurais probablement dû me contenter de mettre le fil bleu sur le fil bleu et le fil rouge sur le fil rouge comme vous me le conseilliez. Cette thèse vous est dédiée.

TABLE DES MATIÈRES

INTRODUCTION GÉNÉRALE

La conteneurisation est une technologie de virtualisation de niveau système d'exploitation permettant de lancer très simplement des logiciels sous forme de conteneurs logiciels, isolés du reste du système. L'utilisation de plus en plus massive de la conteneurisation offre des avantages significatifs par rapport aux approches équivalentes. Tout d'abord, les conteneurs ont des performances significativement meilleures par rapport aux machines virtuelles. Les conteneurs sont également plus simples à déployer en comparaison des approches équivalentes. Enfin, le développement des conteneurs sous forme de microservices facilite leur portabilité et leur réutilisabilité. Les conteneurs sont aujourd'hui utilisés dans de nombreux domaines, allant du cloud multi-tenant, au calcul distribué, en passant par les systèmes industriels.

Toutefois, l'utilisation de conteneurs engendre de nouveaux problèmes de sécurité. En effet, puisque les conteneurs partagent un noyau complet avec l'hôte, ils possèdent une surface d'attaque importante, les rendant cible d'attaques. Par ailleurs, en plus des mauvaises configurations éventuelles des conteneurs qui peuvent engendrer des problèmes de sécurité, de par l'utilisation de logiciels non durcis et le manque fréquent de mises à jour des images conteneurs, les conteneurs sont souvent affectés par des vulnérabilités exploitables, y compris en production, les rendant vulnérables aux attaques. Les attaques depuis les conteneurs peuvent avoir des conséquences importantes. Il est notamment possible à travers un conteneur compromis d'exploiter une vulnérabilité ou une mauvaise configuration pour prendre le contrôle de l'hôte et des autres conteneurs localisés sur le même nœud. Bien que beaucoup de recherche soit effectuée sur la sécurité des conteneurs, leur niveau reste aujourd'hui insuffisant.

Dans le même temps, le fait que le noyau du conteneur soit partagé avec l'hôte permet de profiter d'opportunités nouvelles en termes de sécurité. Il est en théorie possible de définir des politiques de sécurité personnalisées pour des conteneurs comme s'il s'agissait d'ensembles de processus classiques. Si cela implique parfois de résoudre des problèmes théoriques et techniques difficiles, cela permet également d'améliorer significativement leur sécurité. Il est possible d'appliquer des configurations spécifiques aux conteneurs, des règles de sécurité, notamment via les modules de sécurité Linux. Enfin on assiste

aujourd'hui au développement de politiques de sécurité programmables pour conteneurs notamment via le framework eBPF qui constitue un moyen intéressant d'appliquer des politiques de sécurité plus fines. Améliorer la sécurité des conteneurs, et notamment via les approches programmables constitue donc une piste de recherche prometteuse.

Dans le cadre des travaux de cette thèse, nous cherchons à utiliser la programmabilité du noyau pour améliorer la sécurité des conteneurs. Nous mettons en place des politiques de sécurité à grain très fin utilisables y compris par les conteneurs non privilégies pour améliorer leur sécurité. Nous cherchons à mettre en place ces politiques au niveau du noyau afin qu'elles ne soient pas désactivables y compris par des conteneurs compromis. Nous montrons via le développement des frameworks SNAPPY (Safe Namespace and Programmable PolicY) et SecuHub que de telles politiques s'avèrent être à la fois des moyens efficaces pour réduire la surface d'attaques des conteneurs et pour mitiger d'éventuelles vulnérabilités affectant les conteneurs tout en gardant un impact sur les performances très limité. Nos frameworks peuvent aussi être utilisés pour protéger des processus classiques.

En perspective, cette thèse ouvre de nouvelles contributions et pistes de réflexions sur la sécurisation des conteneurs et des autres abstractions logicielles. Nous montrons que le développement d'approches programmables de niveau noyau pourrait permettre l'amélioration de la sécurité de nombreux systèmes, plus largement que dans le cadre de la conteneurisation.

Les contributions principales de cette thèse sont :

- Après avoir présenté un état de l'art à la fois sur les technologies de virtualisation et sur la sécurité logicielle notamment en environnement conteneur, nous présentons une nouvelle taxonomie permettant de classifier les approches de sécurité pour conteneur de niveau noyau.
- Nous présentons le design et l'implémentation de SNAPPY, un nouveau framework permettant aux processus et aux conteneurs, y compris non privilégiés, de mettre en place des politiques de sécurité à grain fin de niveau noyau. Ces politiques sont empilables et peuvent être chargées à chaud pour protéger des conteneurs, permettant ainsi de minimiser leur surface d'attaque et donc d'améliorer leur niveau de sécurité. Nous intégrons également SNAPPY avec la spécification 'runtime' d'OCI (Open Container Initiative) afin de permettre l'application de SNAPPY à l'intégralité du cycle de vie du conteneur.
- Nous présentons le design et l'implémentation de SecuHub, qui permet d'automa-

tiser la distribution et l'installation de politiques de mitigations pour des CVEs (Common Vulnerabilities and Exposures) existantes, applicables à des conteneurs individuels. Nous illustrons également les capacités de mitigation de SecuHub en présentant des politiques de mitigations pour des CVE existantes.

— Nous montrons finalement que grâce à son design, l'utilisation de SNAPPY et de SecuHub n'engendre qu'un surcoût en performance très faible, ce qui rend ces frameworks particulièrement adaptés pour la protection des conteneurs.

Ce mémoire de thèse est organisé comme suit : Ce mémoire est organisé en 7 parties. Le chapitre 1 présente et compare les concepts associés aux différentes approches de virtualisation et les enjeux de sécurité qui y sont liés. Le chapitre 2 commence par présenter les enjeux de sécurité des systèmes et notamment des conteneurs. Il présente également une nouvelle taxonomie des techniques de défense, classe et compare les principales approches de sécurité conteneur de niveau noyau selon cette taxonomie. Le chapitre 3 présente le design et l'implémentation de SNAPPY, un nouveau framework permettant de définir des politiques de sécurité programmables à grain fin de niveau noyau, laissant la possibilité aux conteneurs même non privilégiés d'améliorer leur sécurité. Le chapitre 4 présente un cas d'usage motivant de SNAPPY à travers le framework SecuHub, permettant de définir des politiques de mitigations programmables pour des CVEs existantes, rendant notamment possible une réponse de sécurité rapide quand une nouvelle vulnérabilité est trouvée. Le chapitre 5 évalue les performances de ces deux frameworks, montrant que SNAPPY et SecuHub peuvent être utilisés en conditions réelles avec un impact sur les performances très faible. Finalement, le chapitre 6 dresse les conclusions liées au travaux effectués dans cette thèse et présente les principales pistes de recherche associées.

Chapitre 1

Virtualisation et sécurité

Résumé du chapitre *Les techniques de virtualisation permettent d'isoler différentes abstractions de machines sur le même hôte, avec un impact variable sur les performances du système et sur la facilité de déploiement des domaines virtualisés. Dans ce chapitre, nous présentons les principales formes de virtualisation et nous montrons leur forces et leurs enjeux en termes de sécurité. Nous étudions plus particulièrement la conteneurisation, ses primitives et les logiciels usuels de management des conteneurs, qui sont au cœur des contributions de cette thèse. Nous terminons ce chapitre avec un comparatif des différentes formes de virtualisation.*

Contents

1.1 Virtualisation et machines virtuelles

Un ordinateur est un système complexe comprenant notamment un ou plusieurs processeurs, de la mémoire, des disques, un clavier, une souris, un écran, et beaucoup d'autres systèmes... Si les programmeurs d'applications devaient comprendre en détail le fonctionnement de tous ces dispositifs, il serait quasiment impossible d'écrire du code, et a fortiori du code utilisant de manière optimisée ces composants. Pour cette raison, les ordinateurs offrent une couche d'abstraction permettant d'utiliser de manière simplifiée ces dispositifs : les systèmes d'exploitation [116]. Les systèmes d'exploitation permettent donc aux processus d'accéder de manière simple et optimisée à ces ressources.

Au sein des systèmes d'exploitations, plusieurs processus sont généralement exécutés simultanément. Ces différents logiciels ont besoin d'accéder aux mêmes ressources. Afin de permettre à ces processus de s'exécuter indépendamment les uns des autres, les systèmes d'exploitations mettent en œuvre des mécanismes d'isolation. Grâce à ces techniques, plusieurs processus peuvent utiliser simultanément ou chacun leur tour les mêmes ressources, indépendamment des autres. Cela simplifie largement la coopération de tels processus.

La virtualisation [14] est une technique permettant de créer des abstractions "virtuelles" d'une ressource (mémoire, isolation d'exécution, ...). En ajoutant un niveau d'abstraction entre la ressource physique et la ressource virtuelle, il est possible de présenter à l'application une version abstraite de cette ressource qui sera utilisable indépendamment de l'implémentation réelle de la ressource. Ainsi, le domaine virtualisé peut agir de manière isolée des autres domaines virtualisés ou non. Le virtualiseur a le contrôle sur les éléments à présenter et à cacher à l'interface virtuelle. Puisque la virtualisation permet de ne montrer que des sous-ensembles de la ressource virtualisée, il est possible de créer une vue spécifique au domaine virtualisé et de l'isoler du reste du système. Cette isolation est aussi due au fait qu'il est impossible pour la machine virtuelle d'accéder directement aux ressources physiques : elle doit toujours accéder aux ressources virtualisées, spécifiques à cette unique machine virtuelle.

La virtualisation se base sur trois techniques : 1) le multiplexage, permettant de présenter plusieurs instances virtuelles d'une ressource indépendantes entre elles et basées sur la même ressource sous-jacente afin de partager cette ressource entre les différentes instances virtualisées, 2) l'agrégation, permettant de présenter plusieurs ressources physiques sous forme d'une seule ressource virtuelle et 3) l'émulation, permettant de "simuler" une ressource physique à partir d'une autre ressource compatible. La ressource émulée est

virtuelle.

La virtualisation est une approche utilisée dans de très nombreux domaines. Il est ainsi possible de virtualiser à la fois les ressources systèmes (processeur, mémoire vive, carte réseau, dispositif de stockage...) et l'environnement d'exécution (isolation entre les logiciels, isolation de l'espace utilisateur pour créer des conteneurs, isolation de ressources systèmes pour créer une machine virtuelle complète).

Dans les années 60 le terme de virtualisation désignait les techniques permettant l'isolation mémoire des processus entre eux [2]. Aujourd'hui, le terme de virtualisation désigne généralement la création de *machines virtuelles* au sens large, c'est à dire d'environnements d'exécution qui s'appuient sur des ressources virtuelles fournies par l'hôte (via l'hyperviseur, le moteur de conteneurisation, ...) similaires aux ressources physiques de l'hôte. Dans ce chapitre, nous mettons en italique le terme de *machine virtuelle* au sens large pour ne pas le confondre avec les domaines virtualisés liés à la virtualisation matérielle et à l'émulation. Dans ce mémoire, nous nommons 'virtualisation lourde' les formes de virtualisation appartenant à l'une de ces deux techniques.

Des *machines virtuelles* peuvent être utilisées à des fins très différentes. Les types de machines virtuelles les plus courants sont :

— **Machines virtuelles de langage de programmation** Certains langages, comme le Java ou le C#, utilisent une machine virtuelle (respectivement la JVM (Java Virtual Machine) [120] et le MSCLR (MicroSoft Common Language Runtime) [48]) pour exécuter les programmes compilés sous forme de langage intermédiaire. Ces *machines virtuelles* particulières ne visent qu'à exécuter une seule application et sont donc en dehors du champ d'intérêt de cette thèse.

— **Machines virtuelles lourdes** La virtualisation complète [14], aussi appelée virtualisation lourde permet d'exécuter des machines virtuelles qui se comportent comme des ordinateurs complets. Les ressources d'exécution exposées aux machines virtuelles sont identiques à celles du système physique. Il est donc possible d'y d'installer des systèmes d'exploitation et d'y lancer des programmes comme sur la machine physique. Les machines virtuelles sont totalement isolées entre elles et vis-à-vis de l'hôte, ainsi sauf attaque très spécifique (attaque par canal auxiliaire [107], exploitation d'une faille de l'hyperviseur [92]) il est impossible pour une machine virtuelle de voir l'extérieur de son domaine virtualisé.

— **Machines virtuelles légères** Les machines virtuelles *légères* sont des *machines virtuelles* pour lesquelles seule une partie du système d'exploitation est virtualisée

et une partie du système appartient à l'hôte. Typiquement, l'espace utilisateur est virtualisé alors que le noyau appartient au système hôte. On parle alors de virtualisation de niveau système d'exploitation [16]. Les conteneurs [90] et les sandboxes appartiennent à cette catégorie. La différence entre virtualisation légère et lourde est donc le placement de la frontière de virtualisation [119].

Toutes les formes de virtualisation *système* évoquées ici permettent effectivement d'exécuter du code de manière isolée du reste du système. L'élément de différenciation entre ces différentes techniques est le placement de la frontière de virtualisation, qui impacte directement l'architecture du domaine virtualisé et la manière avec laquelle il communique avec le système hôte. Il est possible de placer la frontière de virtualisation à de nombreux endroits différents comme détaillé dans le reste de ce chapitre. Le placement de la frontière de virtualisation est un enjeu très important pour les raisons suivantes :

— **Performances** Le placement de la frontière de virtualisation a un impact sur le temps d'exécution des *machines virtuelles* puisqu'elle implique plus ou moins d'opérations pour implémenter la virtualisation de l'environnement. Cela a aussi un impact important sur la taille des entités virtualisées puisque cela implique que les *machines virtuelles* doivent intégrer plus ou moins d'éléments système dans la zone virtualisée pour fonctionner correctement. Par exemple, dans le cadre de la virtualisation lourde, la machine virtuelle contient un noyau complet. Ce n'est pas le cas des conteneurs pour lequel seul l'espace utilisateur est virtualisé et le noyau est partagé avec l'hôte. Des comparaisons de performances entre les conteneurs et les machines virtuelles lourdes sont réalisées dans [34] et plus récemment dans [94]. Ces études montrent que les performances des conteneurs se rapprochent de celles de processus natifs dans la plupart des benchmarks, mais que l'utilisation de machines virtuelles lourdes implique généralement une perte de performances non négligeable.

— **Sécurité** Puisque les abstractions permettant la communication entre les domaines virtualisés et l'hôte diffèrent selon la forme de virtualisation utilisée, les différentes formes de virtualisation possèdent des surfaces d'attaque différentes. Ainsi, un conteneur, qui utilise directement le noyau de l'hôte, utilisera pour communiquer avec celui-ci une grande API (Application Programming Interface) et aura donc une surface d'attaque plus importante qu'une machine virtuelle qui ne partage que des ressources d'exécution virtuelles, ce qui représente une ABI (Application Binary Interface) largement plus restreinte. Une étude comparative de la sécurité des

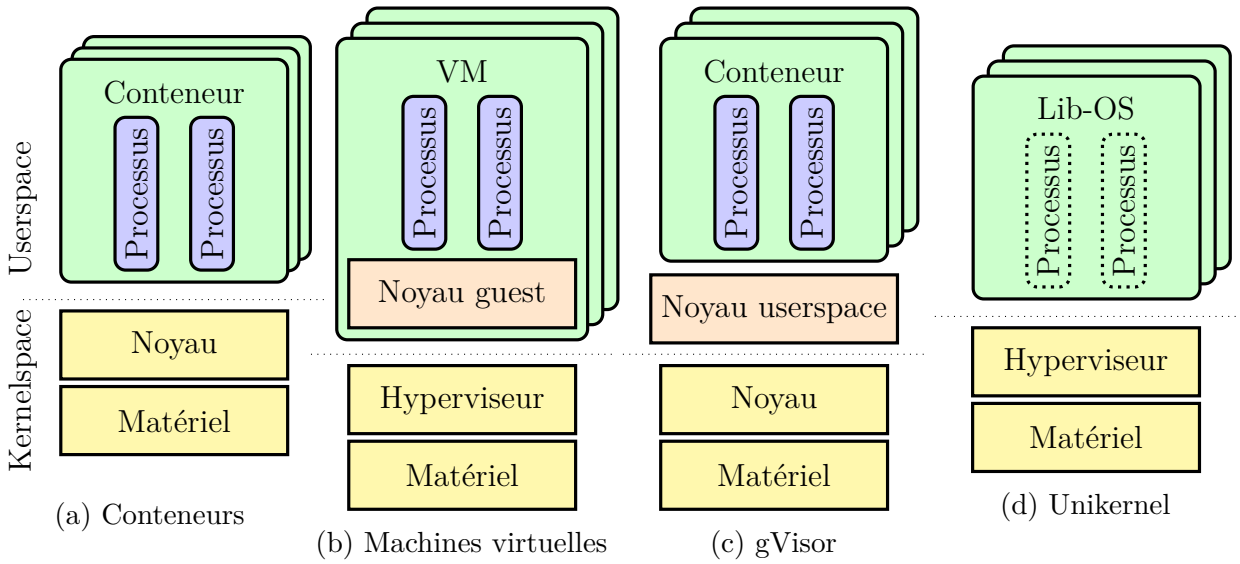
différentes formes de virtualisation est réalisée dans [22]. Cette étude montre que les différents types de virtualisation sont plus ou moins vulnérables aux attaques selon leur type et que les attaques peuvent venir de beaucoup de vecteurs différents. Ces études présentent également les contremesures pouvant être mises en place pour améliorer la sécurité de ces abstractions.

— **Facilité de déploiement** Le niveau de la frontière de virtualisation influe sur la facilité de déploiement de la machine virtuelle. Ainsi, dans le cadre des machines virtuelles lourdes il est nécessaire d'installer complètement l'OS pour mettre en place le système. Toute modification doit être faite depuis l'intérieur de la machine (sauf via des techniques d'introspection ou des connections `ssh`), ce qui peut s'avérer complexe. Au contraire, puisque les conteneurs partagent le noyau avec l'hôte, il est possible de configurer le conteneur finement, y compris en mettant des protections de niveau noyau depuis l'hôte.

— **Actions de sécurité pouvant être mises en place depuis l'hôte** Pour les conteneurs, il est possible de réaliser des vérifications de sécurité depuis l'hôte. Ces protections visent à garantir que le système s'exécute comme prévu. Ces vérifications incluent notamment l'ajout de contrôles d'accès basés sur des règles (RBAC : Rule Based Access Control), la réalisation de contrôle d'intégrité et la mise en place de limites d'utilisation de certaines ressources. Au contraire, puisque le noyau du guest (machine virtuelle) n'est pas censé être accessible/modifiable depuis le kernel hôte (machine physique), ces protections ne sont pas possibles pour des machines virtuelles lourdes depuis le noyau hôte sauf via des techniques d'introspection, imparfaites et coûteuses en performances.

Il est à noter que les techniques de virtualisation ne sont pas exclusives et peuvent dans certains cas être cumulées. Ainsi, l'isolation mémoire des processus entre eux est aujourd'hui appliquée dans tous les systèmes d'exploitation et peut par exemple être cumulée avec la conteneurisation pour assurer un niveau d'isolation supérieur. Similairement, on trouve dans certains environnements où la sécurité est un enjeu majeur, par exemple le NFV (Network Function Virtualization) [125], des conteneurs encapsulés dans des machines virtuelles lourdes pour profiter de la simplicité de déploiement des conteneurs tout en bénéficiant des propriétés de sécurité des machines virtuelles lourdes au prix d'une baisse des performances liée à l'addition de couches logicielles de virtualisation.

Dans ce chapitre, nous présentons les différents mécanismes courants de virtualisation. Nous montrons que la virtualisation permet d'améliorer l'isolation et donc la sécurité de



(a) Conteneurs (b) Machines virtuelles (c) gVisor (d) Unikernel

FIGURE 1.1 – Comparaison architecturale entre les différentes formes de virtualisation

certains systèmes. Nous montrons également qu'au cours des années, la virtualisation s'est généralisée, passant de la simple isolation des processus entre eux jusqu'aux différentes techniques modernes de virtualisation, largement utilisées aujourd'hui. Nous montrons enfin que la virtualisation est associée à des enjeux de sécurité importants.

La figure 1.1 fournit une comparaison architecturale entre les différentes abstractions de virtualisation présentées dans cette section.

Organisation du chapitre Ce chapitre est organisé comme suit : après une présentation de la simple isolation mémoire des processus en 1.2, nous présentons les approches de sandboxing en section 1.3. Nous traitons alors la virtualisation lourde en section 1.4 et la conteneurisation en section 1.5. Nous terminons ce tour d'horizon par une présentation des machines virtuelles légères en section 1.6.1 et des unikernels en section 1.6.2. A la lumière de cette présentation, nous comparons les différentes formes de virtualisation en section 1.7.

1.2 Isolation des processus

Le concept de virtualisation est apparu dans les années 1960, pour permettre l'exécution de plusieurs applications/processus simultanément sur le même ordinateur (mainframe), permettant ainsi de réduire les coûts d'exploitations très importants de ces ma-

chines. Cette virtualisation est réalisée en divisant les ressources systèmes entre les différentes applications. Techniquement, cette isolation repose sur l'utilisation de mémoire virtuelle, permettant l'utilisation par tous les logiciels d'adresses virtuelles et la traduction de ces adresses virtuelles en adresses physiques de la mémoire vive et celle d'espaces d'adressage virtuels.

Ainsi, ces techniques permettent d'interdire à tout processus d'accéder à la mémoire des autres processus. Si un processus se comporte de manière incorrecte ou malveillante, il n'est pas en mesure d'attaquer *directement* les autres processus par exemple en modifiant la valeur d'une adresse mémoire appartenant aux autres processus, améliorant ainsi le niveau de sécurité des systèmes.

Toutefois, même avec l'isolation des processus, tous les logiciels sont en mesure de voir l'intégralité du système (les autres processus exécutés, les systèmes de fichiers, ...). Ce niveau d'isolation reste donc faible et ne suffit pas pour isoler les processus entre eux.

1.3 Sandboxes

Les sandboxes sont un mécanisme de niveau système d'exploitation permettant d'exécuter un processus ou des ensembles de processus dans un environnement système plus restreint. Un tel mécanisme permet notamment d'exécuter du code qui n'est pas de confiance en limitant au maximum ses possibilités, réduisant ainsi les risques d'attaques.

La commande chroot [40] est une première étape de sandboxing, permettant de changer le répertoire racine d'un processus. Ceci est notamment utile pour la création de shells protégés.

Sous Linux, il est possible de restreindre les ressources systèmes visibles (et donc accessibles) à des ensembles de processus à l'aide d'espaces de nommages (namespaces)[42]. Chaque namespace isole une ressource de telle manière à ce que les processus dans les namespaces ne voient la ressource protégée que si elle appartient à la même namespace. Par exemple, un processus dans une namespace PID n'est en mesure de voir que les processus appartenant à la même namespace PID. Cela les isole de fait du reste du système. Le composant namespace, à la base de la conteneurisation est détaillé dans la section 1.5.1.

Plus récemment, les LSM (Linux Security Modules) [75] permettent de créer des sandboxes sous forme de règles. On peut notamment citer SELinux, AppArmor et IMA. L'objectif et le design de ces modules est détaillé plus largement en sous-section 1.5.4.

De nombreux autres systèmes peuvent restreindre l'environnement sur lesquels les

programmes sont exécutés et peuvent donc être considérés comme des sandboxes. C'est notamment le cas de flatpak, windows sandbox, seccomp-bpf, ...

En résumé, les dispositifs de sandboxing peuvent être utilisés individuellement pour restreindre des (ensembles de) processus ou associés entre eux pour créer des environnements très isolés entre eux comme c'est le cas dans le cadre de la conteneurisation, détaillé en section 1.5.

1.4 Virtualisation lourde

1.4.1 Description technique

La virtualisation lourde ou matérielle est un type de virtualisation où les primitives virtualisées sont les ressources système, c'est à dire le processeur la RAM, les I/O... Dans ce cadre, les machines virtuelles sont des systèmes d'exploitation complets comprenant à la fois l'espace noyau et utilisateur.

Il existe plusieurs types de virtualisation lourde :

— **Émulation de machine** Les machines sont implémentées en tant qu'application en espace utilisateur classique et émulent totalement la machine à virtualiser. Puisque la machine doit être totalement interprétée, cela induit une perte de performances considérable. Typiquement, la vitesse d'exécution est réduite par un facteur 5 à 1000 [14].

— **Hyperviseur** Les hyperviseurs permettent une exécution directe sur le CPU pour réduire les pertes de performances. Les hyperviseurs mettent en place un environnement spécifique pour la machine virtuelle pour que celle-ci puisse s'exécuter directement sur le processeur réel. Quand une instruction assembleur doit utiliser une ressource virtuelle, elle est interceptée (trap). L'action à réaliser est alors émulée par l'hyperviseur. Dans les implémentations plus récentes, il est possible de ne trapper que les instructions assembleur sensibles, améliorant ainsi les performances de la machine.

  Un hyperviseur peut s'exécuter directement sur des ressources matérielles (bare metal). On parle alors d'**hyperviseur de type 1**. Au contraire, on peut avoir des hyperviseurs qui s'exécutent au-dessus d'un système d'exploitation. On parle alors d'**hyperviseur de type 2**.

  Les processeurs récents fournissent des extensions liées à la virtualisation (Intel VT-

x [118], AMD-V [3]), permettant ainsi de réaliser matériellement la virtualisation via des instructions assembleur spécifiques (VMCALL, VMLAUNCH, VMWRITE ...) pour exécuter directement toutes les instructions de la machine virtuelle sur le processeur sans avoir besoin de trapper et émuler certaines instructions. On parle alors de **virtualisation matérielle**. La **paravirtualisation** est une autre technique dans laquelle l'hyperviseur réalise des changements dans le système d'exploitation de la machine virtuelle afin de la rendre compatible avec le matériel sous-jacent. La paravirtualisation peut donc être utilisée même sans support matériel de la virtualisation. Enfin, on parle de **virtualisation complète** ou logicielle lorsque l'hyperviseur est exécuté sans support matériel sur des OS non modifiés.

1.4.2 Opportunités et enjeux de sécurité

La virtualisation lourde possède de nouveaux besoins et engendre de nouvelles opportunités en termes de sécurité. Ainsi, si un attaquant tente de compromettre l'hôte du système, après avoir pris le contrôle de la VM, il devra trouver une faille dans l'hyperviseur pour prendre le contrôle du système. Puisque l'hyperviseur fournit une ABI très restreinte, la probabilité pour un attaquant d'y trouver une faille exploitable est faible. On ajoute donc une couche de sécurité par rapport à une exécution utilisant le kernel hôte. De plus, puisque chaque machine dispose de son propre noyau, sa sécurité peut être configurée indépendamment pour correspondre aux besoins spécifiques du domaine virtualisé.

D'un autre côté, les machines virtuelles peuvent être complexes à maintenir. Chaque machine virtuelle doit être mise à jour individuellement (mises à jour de sécurité, changement de configuration, ...). Dans le cadre de parcs contenant un grand nombre de machines ce maintien à jour peut être fastidieux et peut donc ne pas être fait correctement, engendrant des problèmes de sécurité. S'il est possible de contrôler la sécurité des machines via des solutions externes (SIEM, Security Information and Event Management), cette opération n'est pas automatique.

Dans le cas où une machine virtuelle n'est pas de confiance (par exemple utilisée par un client d'un cloud multi-tenant), l'hôte n'a *en théorie* aucun moyen de savoir ce qui se passe dans la machine virtuelle cliente pour protéger son infrastructure. Ceci est dû au gap sémantique : le noyau hôte ne connait pas *a priori* la sémantique de la machine hôte. Il n'est donc pas en mesure de comprendre ce qui se passe à l'intérieur de la machine. Si des techniques d'introspection [51] permettent de reconstruire la sémantique afin

de comprendre ce qui se passe à l'intérieur de la VM, il est possible pour des VM malveillantes d'utiliser diverses techniques d'obfuscation afin d'empêcher l'hôte de franchir le gap sémantique évitant ainsi les éventuelles mesures de surveillance.

1.5 Conteneurisation

La conteneurisation [90] est une forme de virtualisation de niveau OS permettant l'utilisation de plusieurs instances d'espaces utilisateurs isolées entre elles. Comme la virtualisation lourde, la conteneurisation permet de créer des environnements d'exécutions isolés du reste du système, mais contrairement à elle, seul l'espace utilisateur est virtualisé alors que le noyau est partagé avec l'hôte. Les conteneurs interagissent donc avec l'hôte via des envois d'appels systèmes à travers la frontière de virtualisation.

Les conteneurs présentent des propriétés intéressantes par rapport aux machines virtuelles. Ces propriétés expliquent l'adoption massive des conteneurs ces dernières années.

— **Performances :** Puisqu'un conteneur partage le noyau de l'hôte et que les logiciels sont exécutés sans ajout de couche d'abstraction, l'utilisation d'un conteneur n'impacte (quasiment) pas les performances du système. Un conteneur est donc très similaire en performances à un processus classique [34], ce qui est attirant pour les serveurs où la performance est un critère important. Par ailleurs, puisque les conteneurs ne contiennent que les couches logicielles hautes, les conteneurs peuvent avoir une taille très réduite. Ainsi, l'image du conteneur Alpine est conçue pour ne peser qu'environ 5Mo, bien moins que les équivalents machine virtuelle. Enfin, si plusieurs conteneurs utilisent une couche identique, l'hôte ne conserve qu'une seule copie de cette couche, réduisant largement la taille de l'image.

— **Facilité de déploiement :** Les conteneurs peuvent être configurés très simplement en ligne de commande. Des standards de facto comme la spécification de runtime OCI [87] ou le Dockerfile permettent de configurer finement les conteneurs. Il est par exemple possible de contrôler la pile logicielle à inclure dans le conteneur, les namespaces à utiliser, les capabilités du conteneur, les limitations en performances, les ports à ouvrir ou la commande à exécuter au démarrage du conteneur. Ainsi, les conteneurs sont plus simples à configurer (ou à mettre à jour) que des machines virtuelles complètes.

— **Portabilité :** Puisque les conteneurs sont typiquement utilisés pour fournir des microservices selon la règle "un conteneur, un microservice", il est possible de les

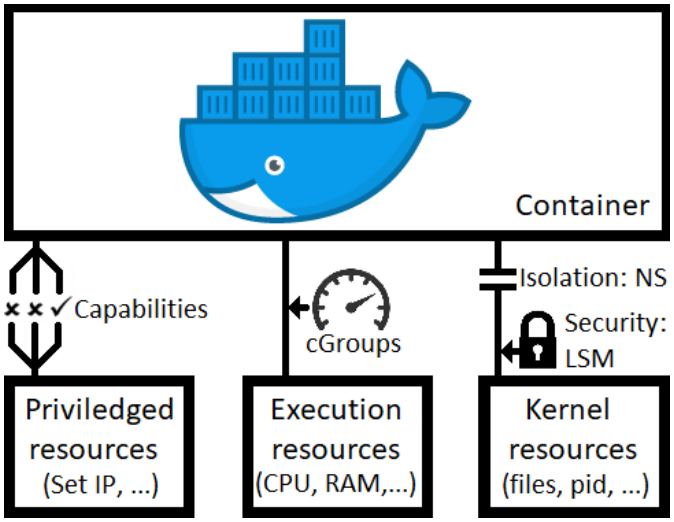


FIGURE 1.2 – Primitives à la base du concept de conteneur

transférer facilement entre machines. Ceci est encore facilité par l'utilisation de dépôts logiciels d'images tels que DockerHub [56]. Par exemple, il est possible de récupérer des images très simplement sur DockerHub avec la commande `docker pull alpine:latest`, qui télécharge la dernière version de l'image Alpine Linux. Si besoin, il est possible de personnaliser cette image à l'aide d'un fichier Dockerfile.

— **Facilité d'orchestration :** Dans le cadre de serveurs hébergeant de nombreux conteneurs, il est possible d'utiliser des orchestrateurs de conteneurs tels que Kubernetes [15] ou Docker Swarm [112]. Ces orchestrateurs automatisent beaucoup de tâches liées aux conteneurs telles que la mise à l'échelle (autoscaling), le déploiement, la surveillance et l'équilibrage de la charge des conteneurs.

Il est à noter qu'au niveau noyau, le concept de conteneur n'existe pas réellement. En réalité, le terme "conteneurisation" désigne l'utilisation coordonnée d'un certain nombre de primitives Linux liées à l'isolation et à la sécurité permettant la mise en place d'environnements d'exécution de type virtualisation légère. Les principales primitives à la base de la conteneurisation sont les namespaces présentées en 1.5.1, les cGroups présentés en 1.5.2, les capabilités présentées en 1.5.3 et les modules de sécurité Linux présentés en 1.5.4. Leur utilisation coordonnée est illustrée par la figure 1.2. Ainsi, on peut considérer que des conteneurs sont *simplement* des sandbox isolant de manière systématique les principales ressources du domaine protégé, du reste du système.

1.5.1 Espaces de nommages Linux

D'après le manuel Linux, Un espace de nommage (namespace) Linux est une abstraction pour la gestion d'une ressource système faisant paraitre aux processus dans les namespaces qu'ils ont leur propre instance isolée de la ressource globale. Les changements opérés sur les ressources dans les namespaces sont visibles dans la namespace mais sont invisibles pour les autres processus [42].

Par exemple, des processus dans une namespace PID (Process IDentifier) ne sont en mesure de voir que les processus situés dans la même namespace PID. Ainsi, si de tels processus tentent de lister les processus système avec `ps -aux`, il leur sera impossible de voir les processus à l'extérieur de leur namespace, les isolant effectivement du reste du système. Ainsi, en mettant un processus dans des namespaces pour toutes les ressources disponibles, il est possible de l'isoler du reste du système. Ce principe est à la base de la conteneurisation, dont l'isolation est gérée intégralement par les namespaces.

La création de namespace peut être réalisée au démarrage d'un processus en utilisant un flag sur les appels systèmes correspondants (`clone()` ou `clone3()`). Il est aussi possible sous certaines conditions de faire changer un processus de namespace avec l'appel système `unshare() ou setns()`. Par exemple, l'appel système `clone(child_fn, stack+STACK_SZ, CLONE_NEWPID|CLONE_NEWNS|SIGCHLD, buf)` crée un nouveau processus dans des nouvelles namespaces PID (`CLONE_NEWPID`) et Mount (`CLONE_NEWNS`). Il est possible de récupérer la namespace du processus `$PID` à l'aide de la commande `lsns` ou de `readlink /proc/$PID/*`.

Les namespaces intégrées dans la version actuelle (v5.13) du noyau Linux sont présentées dans la table 1.1. On constate que la création des namespaces est très ancienne (v 2.4.19, 2002) avec la namespace mount. Les autres namespaces ont été intégrées progressivement dans le noyau et d'autres namespaces pourraient arriver dans le futur. [1]

Certaines namespaces (`pid_namespace` et `user_namesapce`) fonctionnent sous forme d'arbre. Ceci signifie que lorsqu'une nouvelle namespace est créée, elle est considérée comme fille de la namespace du processus à l'origine de sa création. L'objectif de tels arbres est de permettre aux namespaces parent de voir les namespaces filles mais pas l'inverse, permettant ainsi d'avoir une namespace fille plus isolée que la namespace parent.

La figure 1.1 illustre ce comportement pour la namespace PID. Dans cette figure, le PID 4 a été créé dans une nouvelle namespace. Les processus appartenant à cette

1. Il existe une limite sur le nombre de namespace pouvant être créées dans le système puisque la création d'une namespace utilise un flag. Dans l'état actuel du système, seul un flag reste disponible pour la création de nouvelles namespaces [45] rendant l'intégration de nouvelles namespaces dans le noyau difficile. Toutefois, cette limitation technique est levée via l'utilisation de l'appel système `clone3()` [41].

Nom	Ressource isolée	Depuis
Mount	Points de montage système	v 2.4.19
IPC	Queues de messages	v 2.6.19
UTS	Nom d'hôte et nom de domaine	v 2.6.19
PID	Identifiants de processus	v 2.6.24
Network	Matériel réseau, ports réseau	v 2.6.29
User	Identifiant utilisateur et groupe	v 3.8
Cgroups	Dossier racine cgroup	v 4.6
Time	Différentes horloges système	v 5.6

TABLE 1.1 – Liste des namespaces intégrées au noyau Linux

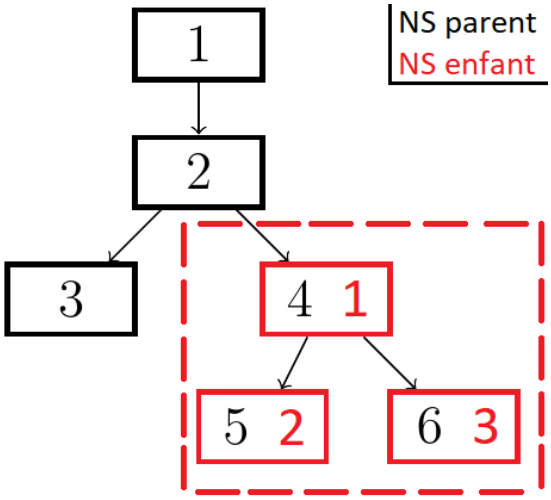


FIGURE 1.3 – Exemple d'arbre de processus et de namespace PID

namespace sont représentés en rouge. Le processus pour lequel la namespace a été créée est considéré comme le PID 1 (rouge). De manière similaire, tous les processus ont un numéro "local" à cette namespace. Les processus dans la nouvelle namespace ne sont pas en mesure de voir les processus dans la namespace parent (noire) mais les processus de la namespace parent sont en mesure de voir les processus de la namespace fille (avec les numéros en noir).

1.5.2 Cgroups

La deuxième brique élémentaire à la base de la conteneurisation est les cgroups (control groups) [39]. Les Cgroups permettent d'isoler et de mettre des limitations sur l'usage des

ressources matérielles (processeur, mémoire, I/O disque, réseau, ...). Dans le cadre des conteneurs les cgroups peuvent donc gérer finement les ressources à allouer aux conteneurs. Cela peut permettre d'éviter que des conteneurs affament le système. Puisque les cgroups ne permettent pas de traiter la sécurité ou l'isolation des conteneurs, nous ne détaillons pas cette fonctionnalité dans cette thèse.

1.5.3 Capabilities

Classiquement, les systèmes de type UNIX/Linux distinguent deux types de processus : les processus privilégiés et les processus non privilégiés. Les processus privilégiés peuvent passer outre toutes les limitations du kernel alors que les processus non privilégiés doivent passer par toutes les vérifications. Un processus peut notamment devenir privilégié à l'aide de `sudo` ou `setuid()`.

Toutefois, cette répartition des privilèges pose des problèmes de sécurité. En effet, il n'est pas possible avec ce système de donner des droits individuels à un processus, par exemple le droit à un processus d'ouvrir un port réseau inférieur à 1024, sans lui donner les privilèges d'administrateur. Ce processus est alors tout puissant et peut réaliser des actions non souhaitées.

Ainsi, les capabilités visent à résoudre ce problème en permettant aux processus d'obtenir des droits particuliers restreints (exemples : `CAP_NET_BIND_SERVICE` : droit d'ouvrir un port réseau, `CAP_AUDIT_WRITE` : droit d'écrire des logs dans le buffer noyau, `CAP_BPF` : employer des opérations eBPF privilégiées) sans donner l'intégralité des droits administrateurs à ce processus.

Dans le cadre de la conteneurisation, il n'est pas rare d'avoir des conteneurs ayant besoin de certains droits particuliers pour fonctionner correctement. Utilisés selon les bonnes pratiques, les conteneurs sont faits pour héberger des microservices, cette liste de droits supplémentaires est donc généralement très limitée. Les capabilités sont donc une solution permettant de réduire les privilèges au minimum en n'autorisant que le strict nécessaire au fonctionnement correct du conteneur. Il est possible de rendre des capabilités héritables ou de les créer uniquement pour certains processus.

1.5.4 Modules de Securité Linux

Les Modules de Sécurité Linux (LSM, Linux Security Modules) [75] sont une fonctionnalité du noyau Linux permettant de mettre en place des modèles de sécurité additionnels

au contrôle d'accès discrétionnaire. Ce dernier permet de limiter l'accès aux ressources en fonction de l'identité du sujet y accédant. Un sujet disposant d'une autorisation est en mesure de la transmettre à d'autres sujets. Le contrôle d'accès discrétionnaire est le contrôle d'accès de base utilisé par Linux. Les modules de sécurité LSM peuvent implémenter plusieurs modèles de sécurité différents tels que du Mandatory Access Control (MAC), du Role-Based Access Control (RBAC) ou de la gestion d'intégrité.

L'architecture des Modules de Sécurité Linux implémente une API présentant aux modules un grand nombre de crochets (de l'anglais hooks) qui sont déclenchés lors d'événements noyau particuliers (ex : ouverture d'un fichier, réception d'un paquet réseau, ...). Il existe un grand nombre de hooks LSM et donc d'événements qui peuvent être surveillés (238 hooks en version 5.13). Divers modules LSM peuvent "enregistrer" ces hooks et donc effectuer des vérifications de sécurité lorsqu'ils sont déclenchés, par exemple quand un fichier est ouvert.

Les événements LSM sont unifiés. Ainsi, un hook LSM se rapporte toujours à un unique événement noyau, indépendant des autres. Ce n'est pas le cas avec l'interception d'appels systèmes. Ainsi, on peut trouver des cas où plusieurs appels systèmes font sensiblement la même opération. Par exemple, les appels systèmes `open()`, `openat()`, `openat2()` et `creat()` peuvent tous être utilisés pour réaliser la même opération. Dans le cas de filtrage par appel système comme avec `seccomp`, il est donc nécessaire de créer des règles pour chacun de ces cas. Inversement, on peut trouver des appels systèmes tels que `prctl` qui peuvent réaliser un grand nombre d'opérations très différentes et peu liées entre elles. Il est donc là aussi nécessaire de faire des règles à l'aide d'un filtrage par appel système. L'approche LSM est donc a priori préférable. car elle permet de traiter unitairement des opérations spécifiques.

Il existe deux types de modules LSM, les modules LSM *mineurs* (par exemple Landlock [103], Yama [43], ...) et *majeurs* (SELinux [111], Apparmor [9], ...). Il n'est possible d'avoir qu'un seul module LSM majeur empilé sur un système. Cette limitation est due au fait que les modules LSM majeurs tentent de contrôler de manière exclusive l'accès aux objets via un champ lié à la sécurité (`f→security`). Les LSM majeurs peuvent également tenter de contrôler de manière exclusive d'autres champs, tels que `secids` ou `cipso`, rendant difficile l'empilage de LSMs. Des travaux sont toutefois effectués au niveau LSM pour permettre d'empiler les LSM majeurs [106]). Il est au contraire possible d'empiler autant de modules LSM mineurs que nécessaire, étant donné qu'ils ne tentent pas de contrôler de manière exclusive des ressources partagées.

Par ailleurs, les LSM sont conçus pour protéger l'intégralité du système. Cela signifie que lorsqu'un module est mis en place, les vérifications qu'il applique sont forcément appliquées à tous les processus du système. Il n'est donc pas possible d'appliquer des LSM uniquement à certains conteneurs. Toutefois, certains modules LSM implémentent en interne des émulations de namespace ou des systèmes de hiérarchie de processus afin de pouvoir appliquer des règles à des sous-ensemble du système. Si ces abstractions permettent effectivement de réaliser des protections par conteneur ou par sous-ensembles du système, nous pensons que cette façon de faire n'est pas optimale. En effet, les protections par hiérarchie de processus ne prennent pas en compte le fait que des processus peuvent être créés à travers des démons systèmes et donc avoir un parent réel différent de leur parent "logique". Par exemple, en démarrant un conteneur avec Docker le parent du conteneur sera `containerd-shim` et le processus l'ayant démarré via la ligne de commande n'appartiendra pas à sa hiérarchie de processus. De tels mécanismes pourraient donc être exploités pour échapper à des protections existantes. Quant aux émulations de namespaces, elles dupliquent un mécanisme noyau largement établi avec une implémentation pour chaque module LSM. Par ailleurs, l'implémentation de ces namespaces émulées est généralement très complexe. Par exemple, la namespace AppArmor repose sur un système de label complexe [38] ce qui peut engendrer des erreurs et des vulnérabilités.

Puisque les modules LSM permettent de mettre en place des règles de sécurité pour l'intégralité du système, leur accès est généralement restreint à l'administrateur. Ceci n'est donc pas adapté aux conteneurs qui ne disposent généralement pas des droits administrateur sur l'hôte pour des raisons de sécurité évidentes. Certains modules LSM comme Landlock [103], permettent toutefois de mettre en place des politiques de sécurité pour protéger leur "périmètre" sans nécessiter de droits administrateur.

Malgré toutes les limitations actuelles des LSM dans le cadre de l'utilisation des conteneurs, les LSM peuvent être utilisés pour améliorer leur sécurité en mettant en place des règles de sécurité personnalisées pour restreindre au minimum les droits des conteneurs et ainsi limiter leur surface d'attaque.

Dans les prochains paragraphes, nous présentons quelques LSMs courants. Il est possible de mettre en place des politiques pour protéger des conteneurs. Par exemple, Docker permet de mettre en place des règles pour ces LSM au moment de la configuration du conteneur.

1.5.4.1 SELinux

SELinux (Security-Enhanced Linux) [111] est le premier module de sécurité Linux à avoir été intégré dans le noyau. Il permet notamment de définir des politiques MAC (Mandatory Access Control) et MLS (MultiLevel Security). Il permet de classer les différentes applications en groupes de sécurité à l'aide d'étiquettes (labels). Ces labels sont appliquées à tous les fichiers directement sur les objets protégés (via `f→security`). Il est alors possible d'appliquer des politiques de sécurité différentes à ces différents sous-groupes.

SELinux est très puissant et possède une granularité très fine. Toutefois, ce module est généralement considéré comme très complexe d'utilisation et il est souvent nécessaire de mettre en place un très grand nombre de règles pour sécuriser un système.

1.5.4.2 AppArmor

AppArmor [9] est un second module de sécurité conçu principalement pour réaliser du MAC. L'objectif d'AppArmor est d'être configurable et maintenable plus simplement que SELinux. Contrairement à ce dernier, Apparmor fonctionne en appliquant des politiques à des chemins et non aux objets directement, ce qui simplifie la mise en place des politiques par rapport au système de labeling de SELinux. Toutefois, certains considèrent qu'une telle protection par chemin est de manière inhérente plus faible que par inode [25].

1.5.4.3 IMA

IMA [100] est un module LSM permettant de mettre en place des politiques d'intégrité. Cela permet de vérifier si des fichiers ont été modifiés que ce soit accidentellement ou intentionnellement. Une telle protection peut compléter les politiques MAC fournies par d'autres modules, dont SELinux ou Apparmor. Il est possible d'intégrer IMA à un Trusted Platform Module (TPM) afin de vérifier l'intégrité du système depuis son démarrage.

1.5.5 Moteurs de conteneurisation usuels

La conteneurisation est aujourd'hui très développée avec de nombreux frameworks très matures. On trouve notamment :

Docker [73] est aujourd'hui le moteur de conteneurisation le plus utilisé en production. En plus de fournir une ligne de commande simple et efficace pour déployer des conteneurs, docker permet le déploiement et la configuration simplifiée d'images à l'aide de Dockerfile.

Ce format de fichier permet d'ajouter à une image existante des fonctionnalités pour l'adapter aux besoins du cas d'usage. Un exemple de Dockerfile est donné en figure 1.1. Dans cette figure on part d'un conteneur `alpine`, on copie un logiciel vers le dossier `/app` du conteneur, on télécharge les logiciels manquants et on compile le logiciel. Le port 12345 est exposé, le dossier maison (`~`) devient `/app` puis le fichier compilé (`./bin`) est exécuté. Cette figure montre donc que la personnalisation d'un tel conteneur peut être réalisé très simplement.

```
FROM alpine:latest
COPY . /app
RUN apk add gcc musl-dev make
RUN make /app
EXPOSE 12345
WORKDIR /app
ENTRYPOINT [./bin]
```

Code 1.1 – Exemple de dockerfile

Les images ainsi déployées utilisent un système de fichier en couches (`overlayfs`). Ainsi, si plusieurs images utilisent la même couche sans modification, par exemple deux conteneurs utilisant la même image de base, les deux conteneurs utilisent la même instance logicielle en mémoire, optimisant ainsi l'utilisation de la mémoire.

En interne, Docker utilise des logiciels de plus bas niveau pour le lancement de conteneurs. La gestion du cycle de vie du conteneur est gérée par le démon système containerd. La gestion à bas-niveau des conteneurs est gérée par runc [88], qui implémente le runtime OCI. Cette séparation entre les logiciels permet un couplage faible entre les différentes briques, une réutilisabilité et la possibilité d'utiliser d'autres briques logicielles en interne (par exemple gVisor [47] à la place de runC).

DockerHub [56], un repository d'images Docker, permet un déploiement très simplifié d'images docker. Il suffit de faire `docker pull alpine:latest` pour télécharger cette image et pouvoir l'exécuter via `docker run`.

Il est à noter que les options de sécurité par défaut de Docker sont souvent imparfaites. Ainsi, par défaut, les conteneurs Docker sont lancés sans user namespace potentiellement en mode privilégié, affaiblissant la sécurité du système.

LXC (Linux Containers) [23] est une alternative à Docker. LXC est généralement utilisé pour faire de la virtualisation de système complet, à différencier de la virtualisation d'application comme réalisé sous docker. LXC est spécifique Linux [32]. LXD (LX Dae-

mon) est une surcouche logicielle à LXC permettant un déploiement simplifié par rapport à LXC.

Le concept de conteneur est très souvent utilisé en environnement Linux. Ce concept existe toutefois également pour d'autres systèmes d'exploitation. Ainsi, les **Solaris Zones** permettent d'exécuter des conteneurs sur Solaris. De plus, Docker fonctionne également sous Mac OS et Windows.

1.5.6 Moteurs d'orchestration usuels

Dans le cadre de la conteneurisation, le déploiement d'applications lourdes peut s'avérer complexe. En effet, cela peut impliquer de déployer de manière coordonnée plusieurs centaines ou milliers de conteneurs différents en prenant en compte des problèmes comme la répartition de charge, l'élasticité de l'infrastructure (autoscaling) ou la gestion des conteneurs. Il est possible d'automatiser ces fonctionnalités à l'aide d'orchestrateurs de conteneurs.

Kubernetes [15] est une plateforme de gestion d'applications conteneurisées. Il permet notamment de réaliser de la répartition de charge de conteneur, y compris entre plusieurs machines. Il permet également de vérifier la "santé" des conteneurs et de tuer et de relancer des conteneurs qui viendraient à dysfonctionner.

Kubernetes est le standard de facto pour l'orchestration. Toutefois, Kubernetes est relativement lourd et son utilisation peut s'avérer complexe.

Docker Swarm [112] est la solution d'orchestration intégrée à Docker. Il est simple d'utilisation en comparaison de Kubernetes mais fournit moins de fonctionnalités et de possibilité d'extensions.

1.5.7 Opportunités et enjeux de sécurité

D'un point de vue de la sécurité, l'utilisation de conteneurs pose des problèmes évidents. Puisque la surface des conteneurs est très élevée, la probabilité d'une attaque liée au moteur de conteneurisation est élevée. Par exemple pour runc, la CVE-2019-5736 [79] permet au conteneurs privilégiés de réécrire arbitrairement sur runc, permettant de prendre le contrôle de l'hôte avec les droits administrateurs. Cela permet aussi de prendre le contrôle de tous les conteneurs situés sur la même machine. Plus récemment la CVE-2021-30465 [82] permet à l'aide d'une attaque par échange de symlink de monter le système de fichiers dans l'hôte, permettant ainsi d'échapper à la conteneurisation. Ce type d'attaque

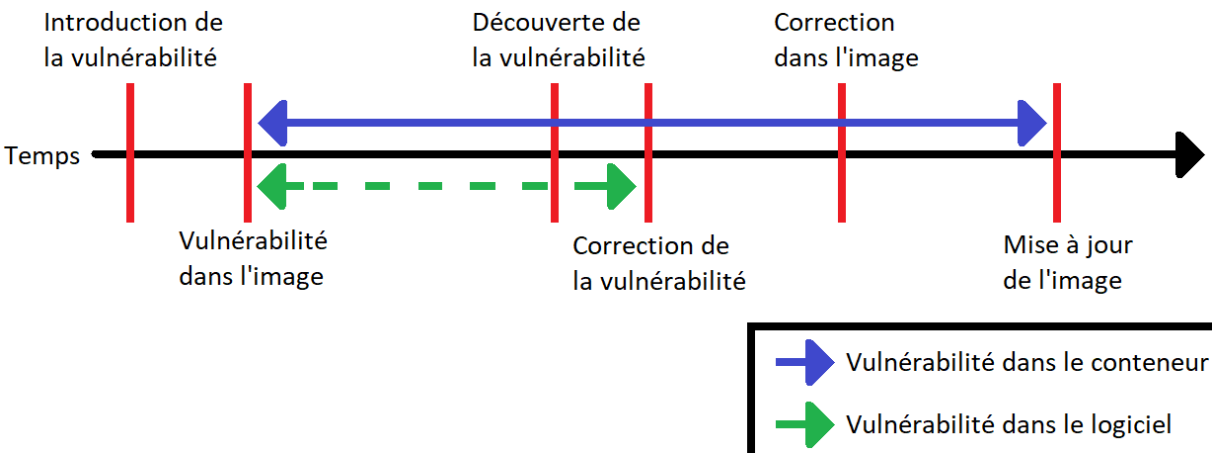


FIGURE 1.4 – Délai de correction des images conteneurs [68]

est donc très dangereux pour les administrateurs de conteneurs.

Par ailleurs, les conteneurs sont souvent exécutés avec de mauvaises options de sécurité. Certains utilisateurs continuent à lancer des conteneurs avec les capabilités d'administrateur (`CAP_SYS_ADMIN`) ou donnent d'autres capabilités trop importantes à leur conteneur, mettant ainsi en danger leur système.

De plus, les images de conteneur sont souvent insuffisamment mises à jour, les laissant vulnérables aux attaques trouvées dans l'intervalle de temps. Ceci est explicable par le fait que les délais se cumulent pour la correction des failles. En effet, lorsqu'une vulnérabilité est découverte, il peut y avoir un délai avant qu'elle ne soit corrigée dans le logiciel. Il y a un second délai avant que le patch soit appliqué dans l'image conteneur et un troisième avant que l'utilisateur de conteneur mette à jour son image vers la version patchée laissant le conteneur vulnérable entre temps [109]. Ces délais sont montrés dans la figure 1.4. Ce problème est encore plus important lorsque les images viennent de dépôts (repositories) non contrôlés comme DockerHub. En effet, en plus des quelques images malveillantes, ces images sont souvent très loin d'être à jour résultant en des vulnérabilités importantes. Ainsi quand une vulnérabilité logicielle est trouvée, il se passe en moyenne 422 jours entre le temps où un patch est rendu disponible et le moment où le développeur met à jour son image vers une version où la vulnérabilité est patchée [68]. Ainsi, plus de 80% des images présentes sur DockerHub contiennent au moins une vulnérabilité classé "High" posant un important risque de sécurité [109].

Afin de pallier les problèmes de sécurité des conteneurs, beaucoup de recherche est réalisée sur des approches intermédiaires entre les conteneurs et les machines virtuelles.

En entreprise, les conteneurs sensibles sont souvent encapsulés dans des machines virtuelles pour les mêmes raisons. Nous présentons certaines de ces approches dans 1.6. Ces approches sont intéressantes en termes de sécurité et peuvent donc constituer une solution pragmatique utilisable à court terme mais elles impliquent un compromis en termes de performance et de taille des images.

Pour résumer, la recherche en sécurité des conteneurs est un domaine de recherche prometteuse. Nous avons montré en quoi les conteneurs possédaient des avantages importants sur les machines virtuelles mais conservaient aujourd'hui un niveau de sécurité intrinsèquement plus faible. Ces dernières années, le niveau de sécurité des conteneurs a déjà été augmenté avec la meilleure prise en compte de la sécurité dans les moteurs de conteneurisation et l'adaptation progressive des modules de sécurité Linux aux conteneurs. C'est pourquoi nous tentons dans cette thèse d'améliorer encore la sécurité des conteneurs pour se diriger vers le niveau proposé par les VMs sans sacrifier les bonnes propriétés des conteneurs.

Il est important de comprendre que puisque les conteneurs ne sont pas différents d'ensembles de processus classiques du point de vue du noyau, il est possible de les protéger comme des processus classiques. De par le placement de la frontière de virtualisation, il est possible de protéger des conteneurs depuis le noyau. Ceci est d'autant plus facile qu'il n'existe pas de gap sémantique dans le cadre des conteneurs. Il est donc possible d'améliorer la sécurité des conteneurs de manière très fine depuis le noyau. Ainsi les conteneurs fournissent une opportunité de développer des services de sécurité innovants afin d'améliorer leur sécurité. Cette opportunité n'est pas offerte par les machines virtuelles. Comme montré en chapitre 2, l'approche permettant de rendre ces services de sécurité les plus flexibles possible est de les rendre programmables. Nous présentons dans cette thèse le design et l'implémentation de tels mécanismes avec SNAPPY présenté en chapitre 3 et SecuHub présenté en chapitre 4.

1.6 Autres formes de virtualisation

Bien que les formes de virtualisation les plus utilisées actuellement soient la virtualisation lourde et la conteneurisation, d'autres formes de virtualisation existent, réalisant des choix différents de placement de la frontière de virtualisation. Nous présentons quelques approches intéressantes dans cette section.

1.6.1 Machines virtuelles légères

Plusieurs frameworks proposent l'implémentation de machines virtuelles très légères conçues pour limiter leur impact en performances.

KataContainers [99] est un framework permettant d'utiliser des environnements utilisables comme des conteneurs et compatible avec la spécification OCI mais construits en réalité comme des machines virtuelles légères. Une telle abstraction permet d'obtenir la facilité de déploiement des conteneurs et les propriétés d'isolation des machines virtuelles au prix d'une perte de performances liée à la virtualisation et à la taille augmentée des conteneurs liée au noyau guest. De par les optimisations réalisées, Katacontainers est plus efficace qu'une machine virtuelle classique mais reste à des niveaux inférieurs à des conteneurs natifs [64]. Finalement, KataContainers supporte moins d'options de sécurité que runc.

FireCracker [1] est un autre framework implémentant un VMM (moniteur de machines virtuelles) qui s'intègre à KVM (Kernel-based Virtual Machine) [62]. Ce VMM vise à être le plus minimaliste possible et supprime donc toutes les fonctionnalités secondaires (support pour l'USB, le PCI, l'émulation d'instruction CPU, ...), réduisant le coût en performances de l'utilisation de machines virtuelles. FireCracker est notamment utilisé dans Amazon AWS.

SCONE [7] permet de déporter l'exécution de code critique d'un conteneur, par exemple une signature cryptographique, vers une enclave Intel Software Guard eXtension (SGX) afin de préserver la confidentialité du conteneur contre un hôte malveillant. Toutefois, SCONE engendre un surcoût en performances très important le rendant inutilisable en production et requiert des modifications logicielles non-triviales. Enfin, à cause de vulnérabilités critiques trouvées dans SGX, les opérateurs cloud pourraient vouloir désactiver SGX, désactivant incidemment SCONE.

gVisor [47] fournit une autre abstraction de machines virtuelles légères visant à être utilisées comme des conteneurs. Pour réduire la surface d'attaque des conteneurs (un noyau Linux complet), gVisor développe un noyau léger exécuté en espace utilisateur au-dessus d'hyperviseurs. Ce noyau léger, implémenté en Go réduit le nombre d'appels systèmes exposés au conteneur en présentant 55 appels système qui permettent d'en implémenter 211, réduisant ainsi la surface d'attaque de tels conteneurs. gVisor n'implémente pas tous les appels système et n'est donc pas compatible avec les applications utilisant des appels systèmes non supportés. Si un tel co-noyau permet d'améliorer la sécurité des conteneurs et le Go est connu pour avoir de meilleures propriétés de sécurité que le C, il

implique un surcoût en performances assez important. Ainsi, en comparaison de moteurs de conteneurisations classiques, les appels systèmes sont 2.2 fois plus lents, les allocations mémoires 2.5 fois plus lentes et l'ouverture de fichiers 216 fois plus lente [126]. Puisque gVisor implémente le runtime OCI, le standard de facto de la conteneurisation, il peut s'intégrer en tant qu'outil de bas niveau avec les moteurs de conteneurisation comme Docker, remplaçant alors runc.

1.6.2 Unikernel

Les unikernels sont des images spécialisées pour des applications spécifiques. Contrairement à des noyaux classiques, les unikernels ne possèdent qu'un seul espace d'adressage mémoire. L'application exécutée avec un unikernel embarque directement les dépendances minimales correspondant aux constructions OS nécessaire à l'exécution de l'application sur la machine.

De par l'absence de besoin d'isolation mémoire, il est possible d'exécuter les applications sans transition de privilège, résultant en des temps de boot et d'exécution plus courts. Cette absence d'isolation mémoire peut toutefois poser des problèmes de sécurité lorsque l'isolation entre les processus est souhaitable (par exemple pour un serveur ssh) et il est complexe de réaliser l'isolation de LibOSes tentant d'accéder aux mêmes ressources. Il est possible d'exécuter des Unikernels au-dessus d'hyperviseurs permettant d'exécuter de multiples applications sur la même machine au prix d'un surcoût en performances lié à la virtualisation. Bien que les Unikernels soient une ligne de recherche largement explorée, ils restent très marginalement utilisés en production. Un problème récurrent avec les unikernels est la compatibilité logicielle puisqu'il est généralement nécessaire de réécrire les applications. C'est notamment le cas pour Mirage OS, pour lequel les applications doivent être réécrites en Go pour pouvoir être exécutées. LightVM [71] nécessite aussi du temps de développement pour rendre l'image compatible Unikernel et personnaliser les paramètres de l'image pour optimiser ses performances. D'autres unikernels comme OS^v nécessitent que l'application soit recompilée spécifiquement en tant que relocatable Shared Object. Finalement, d'autres unikernels plus récents tels que HermiTux permettent de conserver une compatibilité binaire avec les distributions classiques et simplifie donc le déploiement de telles primitives.

X-Containers[108] est une autre approche pour mettre en place de la virtualisation légère (donc pas des conteneurs contrairement à ce que le nom laisse penser) à partir d'un unikernel. X-Containers permet de construire des LibOS s'exécutant au-dessus d'un noyau

	Processus	Sandbox	Conteneur	VM lourde	VM légère	Unikernel
Isolation	Mauvais	Moyen	Bon	Bon	Bon	Bon
Isolation mém.	Bon	Bon	Bon	Bon	Bon	Mauvais
Performances	Bon	Bon	Bon	Mauvais	Moyen	Bon
Util. mém.	Bon	Bon	Bon	Mauvais	Moyen	Moyen
Surf. d'attaque.	Mauvais	Mauvais	Mauvais	Bon	Bon	Bon
Facilité d'util.	Bon	Bon	Bon	Moyen	Moyen	Mauvais

TABLE 1.2 – Comparaison des principales formes de virtualisation

Xen. X-Containers utilise moins d'appels système qu'un noyau Linux classique réduisant l'impact en performances et la surface d'attaque par rapport à des conteneurs classsiques. Cette approche induit par design un temps de boot plus long que les conteneurs et une efficacité mémoire plus faible principalement due à des surcoûts pour les opérations sur les tables de pages mémoire.

1.7 Comparaison des différentes formes de virtualisation

Il existe aujourd'hui de nombreuses formes de virtualisation. Ces abstractions réalisent des choix de placement de la frontière de virtualisation différents. Ces choix ont des impacts importants sur la performance, la facilité de déploiement, le niveau d'isolation, la surface d'attaque, le rôle du domaine virtualisé et les capacités du domaine virtualisé en termes de sécurité.

La table 1.2 résume les avantages et limitations des différentes formes de virtualisation. Chacune de ces formes de virtualisation implique un compromis différent selon ces divers indicateurs et aucune forme de virtualisation n'est strictement supérieure aux autres. En termes d'isolation mutuelle, les processus ne sont pas isolés entre eux, quant aux sandboxes, leur isolation reste faible. L'isolation mémoire fonctionne pour toutes ces abstractions sauf les unikernels qui par design partagent un même espace mémoire. En termes de performances, les machines virtuelles ont sont moins avantageuses de par les couches d'abstractions supplémentaires liées aux deux noyaux. Les machines virtuelles légères tentent de limiter ces impacts mais ces impacts restent supérieurs à ceux des autres abstractions. En termes d'utilisation mémoire, les machines virtuelles ont aussi un impact significatif lié au stockage du noyau et à la duplication des pages mémoire quand plusieurs

machines virtuelles similaires s'exécutent sur un hôte. En termes de surface d'attaque, il est possible d'utiliser toutes les abstractions de l'hôte (dont un noyau complet) depuis un processus classique, une sandbox ou un conteneur. Leur surface d'attaque est donc importante. En termes de simplicité d'utilisation, la gestion d'une machine virtuelle peut être relativement complexe du fait du besoin d'installer et de maintenir à jour la VM en plus de l'hôte. Les Unikernels sont généralement très complexes à utiliser et du travail spécifique est généralement nécessaire pour compiler et optimiser une LibOS.

En conclusion, les différentes formes de virtualisation présentées dans ce chapitre proposent des compromis différents en termes de performances, de facilité de déploiement et de sécurité. Elles sont par ailleurs plus ou moins adaptées à la mise en place de politiques de sécurité de niveau noyau.

Dans ce contexte, les conteneurs sont une forme de virtualisation intéressante, de par leur bonnes propriétés notamment en termes de performances, de facilité d'utilisation et de déploiement. Si les conteneurs posent de nouveaux enjeux de sécurité, leur partage de noyau avec l'hôte permet de mettre en place des protections spécifiques innovantes directement depuis le noyau de l'hôte. Les approches proposées dans les chapitres 3 et 4 s'inscriront plus en profondeur dans cette ligne de recherche.

Dans la prochaine section, nous traiterons des problématiques de sécurité subies par les systèmes, et particulièrement en environnement conteneurs. Nous aborderons d'abord les problématiques de sécurité du côté offensif puis les moyens de défenses contre ces problématiques. Nous présenterons notamment une nouvelle taxonomie des défenses de niveau noyau pour les conteneurs et classerons les principaux frameworks de défenses de niveau noyau applicables aux conteneurs selon cette taxonomie.

Chapitre 2

Défenses contre les attaques

Résumé du chapitre *La sécurité est en enjeu important pour les conteneurs du fait de leur utilisation massive en production et de leurs faiblesses intrinsèques en matière de sécurité. Ce chapitre donne un résumé des attaques historiques, de leur évolution et des contremesures possibles puis nous étudions plus particulièrement les spécificités des conteneurs en matière d'attaques et de traitement de celles-ci. Nous détaillons les principales formes de défense applicables aux conteneurs, au travers d'une nouvelle taxonomie que nous utilisons pour analyser l'état de l'art. Nous terminons ce chapitre en montrant en quoi les récentes avancées en matière de programmabilité du noyau ouvrent de nouvelles voies pour améliorer la sécurité des conteneurs*

Contents

2.1 Introduction : les grands enjeux de la sécurité

2.1.1 Attaques informatiques : approche historique

La sécurité informatique est un enjeu relativement récent à l'échelle de l'informatique. Aux prémices de l'informatique, les acteurs pouvant utiliser des systèmes informatiques étaient très peu nombreux, disposaient de connaissances en informatique avancées et devaient accéder physiquement aux machines. L'arrivée des réseaux informatiques, dont Arpanet en 1969, puis Internet, apparu au cours des années 1980 et démocratisé dans les années 1990/2000 via le web a rendu les machines connectées. Il est alors possible de dialoguer et de contrôler des machines à distance. Cette interconnexion des machines a augmenté les capacités des machines mais les a rendues plus vulnérables.

Ainsi, la première attaque informatique ayant fait des dommages significatifs est le ver Morris (1988) [89]. Ce ver ne cherchait pas initialement à causer de dommage mais simplement à se propager pour mesurer la taille du réseau. Toutefois, en surchargeant le processeur des victimes, il a causé des dénis de services importants. Ce ver a infecté 6000 ordinateurs soit environ 10% du parc de machines de l'époque.

La généralisation progressive d'Internet pour le grand public dans les années 1990 et 2000 permise par l'apparition du WorldWide Web (www), des navigateurs web tels que Netscape ou Internet Explorer et des moteurs de recherche dont Yahoo et Google a largement augmenté les enjeux de sécurité. Plus de machines étaient alors connectées à Internet, dont un grand nombre d'utilisateurs n'ayant aucune connaissance en informatique. Cela a généré une augmentation importante du nombre d'attaques. On peut notamment citer l'exemple du ver Mélissa (1999) ou du virus ILoveYou (2000).

Depuis les années 2000, la démocratisation d'internet et particulièrement auprès d'utilisateurs n'ayant aucune connaissance en informatique et la mise en place de services lourds a considérablement augmenté la surface d'attaque des systèmes. Ainsi, le nombre de machines connectées à Internet est passé d'environ 250 millions d'utilisateurs au tournant du millénaire à environ 1,8 milliard début 2010 et dépasse les 5 milliards d'utilisateurs aujourd'hui, soit environ deux tiers de la population mondiale [123].

Cette évolution a coïncidé avec le développement de services aux enjeux de sécurité importants tels que le e-commerce puis le m-commerce. De plus en plus de machines hétérogènes se sont connectées à Internet, telles que des téléphones, tablettes, systèmes informatiques industriels et tout types d'objets connectés. Bien que très différents, ces systèmes possèdent tous des enjeux de sécurité importants.

Par exemple, les dispositifs industriels gèrent des systèmes critiques et toute attaque peut avoir des conséquences dramatiques. Il est à noter que les dispositifs industriels critiques ne sont généralement pas *directement* connectés à Internet mais utilisent un réseau OT (Operational Technology) dédié. De nombreuses attaques ont toutefois été réalisées, souvent en attaquant d'abord le réseau IT puis se propageant au réseau OT. On peut notamment citer l'attaque Stuxnet [33] visant le réseau nucléaire Iranien ou les ransomware (par exemple WannaCry [19]) touchant particulièrement ce genre de systèmes.

De l'autre côté du spectre, les objets connectés revêtent des enjeux de sécurité plus limités mais possèdent souvent des niveaux de sécurité très faibles (mots de passe en dur, ports ouverts, utilisation de logiciels très vulnérables et non mis à jour). Ils forment donc aussi une cible de choix pour les attaquants. Puisque ces objets sont très nombreux, il est possible de compromettre un grand nombre de machines avec une seule attaque. Ainsi le malware Mirai a pris le contrôle d'un grand nombre de caméras connectées et s'en est notamment servi pour réaliser un DDOS contre les serveurs de l'hébergeur OVH en 2016.

Ainsi au cours des années, les enjeux de sécurité sont devenus de plus en plus importants et ils constituent aujourd'hui un enjeu primordial.

Dans le reste de cette section, nous verrons quelles sont les principales attaques et quelles conséquences elles peuvent avoir. Nous verrons ensuite comment sont gérées les attaques aujourd'hui de leur découverte à leur correction.

2.1.2 Types d'Attaques et conséquences

La sécurité est un champ d'application très large. Il existe un très grand nombre de catégories de vulnérabilités.

L'OWASP (Open Web Application Security Project) [44] fournit régulièrement un classement des types de vulnérabilités les plus critiques. Ce classement permet de se faire une idée des principaux enjeux de sécurité. Le classement OWASP 2017 est présenté dans le tableau 2.1. On constate que les failles de sécurité peuvent exploiter des vecteurs d'attaques de types très différents.

La défense contre les attaques est donc un champ d'action très vaste puisque des failles très différentes peuvent être découvertes à des endroits très divers de systèmes toujours plus complexes et toujours plus connectés.

#	Type	Description
1	Injection	Exécution / interprétation malveillante de données non trustées
2	Broken Authentification	Implémentation incorrecte de fonctions d'authentification
3	Sensitive Data Exposure	Protection incorrecte de données sensibles
4	XML External Entities	Mauvaise configuration XML prenant en compte des ressources externes
5	Broken Access Control	Mauvaises autorisations d'accès
6	Security Misconfiguration	Configuration incorrecte
7	Cross site Scripting	Exécution de script utilisateur sans validation ou échappement correct
8	Insecure deserialization	Désérialisation incorrecte
9	Using Known Vulns.	CVE existantes mais pas corrigées
10	Bad Logging/Monitoring	Pas assez de surveillance du système

TABLE 2.1 – Top 10 OWASP

2.1.3 Divulgation et correction des failles

Lorsqu'une vulnérabilité est trouvée, plusieurs méthodes de divulgation coexistent [104] :

— **Divulgation coordonnée** : Le découvreur de la vulnérabilité transmet les informations de la vulnérabilité à corriger. L'entreprise a alors un délai pour corriger la vulnérabilité dans ses produits affectés. La vulnérabilité n'est révélée au grand public que lorsqu'elle est effectivement corrigée. Cela permet de limiter le risque que des pirates exploitent la vulnérabilité mais peut rajouter du délai avant la correction de faille. Dans certains cas, l'entreprise peut mettre beaucoup de temps à corriger la faille voire ne pas souhaiter corriger du tout la vulnérabilité, laissant les logiciels vulnérables.

— **Divulgation complète** : Le découvreur de la vulnérabilité transmet le plus rapidement possible toutes les informations nécessaires à l'exploitation. Cette approche permet aux administrateurs système de corriger plus facilement la vulnérabilité mais permet aussi à d'éventuels pirates d'exploiter la vulnérabilité beaucoup plus simplement. Dans certains cas, cette divulgation n'est effectuée que vis-à-vis d'un groupe restreint d'individus, on parle alors de **divulgation limitée**.

— **Non-divulgation**. Cette technique consiste à ne pas divulguer la vulnérabilité permettant au découvreur ou à d'autres pirates d'attaquer le système. Cette technique affaiblit généralement la sécurité du système et est à éviter.

Ainsi, dans le cas de la divulgation coordonnée, le scénario de découverte d'une attaque est le suivant : quand une vulnérabilité est trouvée dans un logiciel, la personne ayant découvert la vulnérabilité contacte le développeur du logiciel ainsi qu'un CNA (CVE Numbering Authority). Le développeur a alors une période de temps pour prendre des mesures appropriées, comme le développement d'un patch pour le logiciel vulnérable. Après un certain temps, un numéro unique de CVE est attribué à cette vulnérabilité. La vulnérabilité est alors rendue publique sous ce numéro de CVE. Le rapport contient notamment une description technique détaillée des conditions de l'exploit. Dans certains cas, le développeur fournit aussi une preuve de concept de l'exploit. Dans la majorité des cas, un patch est rendu disponible avant que la vulnérabilité soit révélée. La vulnérabilité reste toutefois dangereuse. En effet, le patch n'est généralement pas immédiatement appliqué dans tous les environnements utilisant le logiciel vulnérable. Par ailleurs, les utilisateurs ne font pas toujours leurs mises à jour régulièrement, laissant leur environnement vulnérable quand une CVE est trouvée. De plus, les patchs très récents contiennent régulièrement des bugs, pouvant soit laisser des chemins d'exploitation de la vulnérabilité, soit créer des comportements inattendus voire même générer d'autres vulnérabilités. Enfin, la correction d'une vulnérabilité logicielle nécessite au moins de redémarrer le logiciel et parfois la machine. Ceci n'est pas possible dans les environnements nécessitant une haute disponibilité. Le patch n'est donc pas une solution suffisante dans tous les cas pour corriger des vulnérabilités et des solutions de mitigation peuvent constituer des alternatives ou des compléments avantageux au patch dans certains cas.

Contributions de l'auteur sur cette thématique

- Leveraging Kernel Security Mechanisms to Improve Container Security : a Survey. **Maxime Bélair**, Sylvie Laniepce, and Jean-Marc Menaud. In Proceedings of the 14th International Conference on Availability, Reliability and Security (ARES '19). Août 2019. Canterbury, Royaume-Uni DOI : `https://doi.org/10.1145/3339252.3340502` [11]
- Container interaction with host OS for enhanced security : a survey. **Maxime Bélair**, Sylvie Laniepce, Jean-Marc Menaud. Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS 2019), Juin 2019, Anglet, France

Organisation du chapitre Le reste de ce chapitre est organisé comme suit : nous commençons par présenter les contremesures contre les attaques en section 2.2 puis nous

présentons les failles auxquelles sont confrontés les conteneurs en section 2.3. Nous présentons alors une nouvelle taxonomie des approches défensives de niveau noyau en section 2.4. Nous classons alors les principaux frameworks de défense de niveau noyau selon cette taxonomie : les configurations sont présentées en section 2.5, les approches basées sur le code en 2.6 et les approches basées sur les règles en section 2.7. Nous comparons ensuite ces approches en section 2.8 et nous terminons ce chapitre en montrant comment la programmabilité du noyau peut être un vecteur d'amélioration de la sécurité système et conteneurs en section 2.9.

2.2 Contremesures contre les attaques

D'un point de vue de la défense, il existe plusieurs moyens de lutter contre les attaques, détaillés dans cette section. Ces approches possèdent des avantages et des inconvénients. Il est important de comprendre qu'aucune de ces approches n'est parfaite et qu'un système sécurisé utilise plusieurs de ces techniques.

2.2.1 Correctif de sécurité

L'approche la plus classique pour corriger une vulnérabilité est d'appliquer un correctif de sécurité (patch) pour celle-ci. Cela implique pour des développeurs de préparer un patch pour la vulnérabilité. L'utilisateur du logiciel vulnérable peut alors mettre à jour son logiciel vers une version corrigée. Cette approche est largement utilisée et un maintien régulier des logiciels à jour permet de restreindre largement les risques d'attaques. Cette approche possède toutefois un certain nombre de limitations :

— Lorsqu'une vulnérabilité est trouvée, un correctif logiciel n'est pas toujours disponible.
— Les correctifs, notamment les plus récents peuvent ne corriger que partiellement une vulnérabilité, peuvent être complexes et donc difficile à auditer et peuvent rajouter de la surface d'attaque.
— Certains correctifs peuvent nécessiter des vérifications complexes ou des modifications algorithmiques majeures et donc réduire les performances du système. C'est par exemple le cas des premiers correctifs logiciels pour les vulnérabilités Spectre.

2.2.2 Réduction de privilèges

La réduction de privilèges est une technique visant à restreindre les actions autorisées pour un (ensemble de) processus ou de logiciel(s). Cette approche ne vise pas à protéger d'une faille en particulier mais à éliminer le risque que des fonctionnalités non nécessaires au fonctionnement du système génèrent des failles de sécurité exploitables. De nombreuses approches peuvent permettre de réduire l'ensemble des opérations admissibles et peuvent donc servir à la réduction des privilèges, dont les capabilités, les namespaces, les Linux Security Modules et la virtualisation. Toutes ces approches ont déjà été présentées dans le chapitre 1. En plus de ces approches de niveau noyau, des approches de niveau espace utilisateur peuvent être mises en place, par exemple par la mise en place d'une API sécurisée. Toutefois, ces approches peuvent être contournées par un utilisateur malveillant par exemple via l'utilisation directe de l'API non sécurisée.

Ainsi, la réduction de privilèges permet effectivement de limiter les risques d'attaques et est donc largement utilisée en production mais possède un certain nombre de limitations présentées ci-dessous :

— La réduction de privilèges s'appuie sur la désactivation de comportements plus ou moins finement. Dans certains cas il n'est pas possible de désactiver assez précisément un comportement qui peut être dangereux. C'est notamment le cas quand des fonctionnalité légitimes ont un fonctionnement proche de fonctionnalités à désactiver.
— Dans certains cas, l'environnement d'exécution peut être fourni sous forme de boite noire. C'est notamment le cas dans certains environnements NFV (Network Functions Virtualization) et cloud multi-tenant. Il est alors difficile pour l'infrastructure de déduire précisément le comportement attendu et il est de fait difficile de réduire largement la surface d'attaque de l'environnement ; seules des politiques génériques peuvent être mises en place.
— Dans certains cas, l'application de politiques de sécurité incorrectes peuvent bloquer des comportements légitimes mais rares, par exemple une levée d'alerte. Les systèmes affectés fonctionnent alors de manière incorrecte avec des conséquences potentiellement graves en production (pannes, comportement incorrect, ...). Les approches visant à automatiser la génération de politiques de sécurité sont particulièrement vulnérables à ce problème.

2.2.3 Mitigation spécifique

La mitigation spécifique de vulnérabilités est une approche complémentaire à la correction de failles. Elle permet d'empêcher l'exploitation d'une faille spécifique sans modification du logiciel affecté. La mitigation consiste en l'ajout d'une politique de sécurité conçue pour empêcher *spécifiquement* cette vulnérabilité. Il est là aussi possible de mettre en place de telles politiques de sécurité à la fois depuis l'espace utilisateur ou depuis le noyau, bien que les approches de niveau noyau, telles qu'explorées dans cette thèse soient préférables car les politiques sont alors stockées dans un espace mémoire différent des processus qu'elles protègent. Puisque les conteneurs sont parfois malveillants, cette propriété garantit que les politiques ne sont pas désactivables, même en cas de compromission du conteneur.

Dans beaucoup de cas, dont ceux explorés dans cette thèse, il est possible d'écrire une mitigation pour une vulnérabilité plus simplement qu'un correctif, de par le fait qu'il est possible d'exécuter une règle ou un morceau de code à un endroit précis, où le vecteur d'attaque de la vulnérabilité sera plus simple à détecter et à bloquer que depuis le logiciel. La mitigation spécifique peut servir dans l'attente d'un correctif ou un complément d'un correctif manquant de maturité. Une fois la stabilité du correctif garantie sur le long terme, il est possible de désactiver la politique de mitigation.

Cette approche possède toutefois des limites :

— Mitiger certaines vulnérabilités complexes peut s'avérer difficile voire impossible au niveau kernel. Par exemple, une vulnérabilité exploitée purement en espace utilisateur comme un buffer overflow ne peut pas être détectée directement depuis l'espace noyau et constitue donc un pire cas pour la mitigation de niveau noyau.
— Dans le cas de systèmes très vulnérables, la mitigation de toutes ses vulnérabilités peut nécessiter beaucoup de politiques différentes.

2.3 Failles conteneurs

Dans cette thèse, nous nous intéressons particulièrement aux environnements conteneurs. Nous présentons donc dans cette section la spécificité des vulnérabilités affectant les conteneurs par rapport aux autres environnements. La sécurité des conteneurs est plus largement discutée dans [113] et les mécanismes de défense dans [114].

2.3.1 Mauvaise configuration du conteneur

Un vecteur majeur d'attaques est la configuration incorrecte du conteneur, pouvant avoir des conséquences majeures. Les cas les plus classiques sont :

Droits non nécessaires On peut trouver certains environnements en production où les conteneurs sont lancés en tant qu'administrateur sans namespace utilisateur, diminuant largement leur niveau de sécurité. Similairement, en donnant des capabilités non nécessaires aux conteneurs, on leur donne inutilement des droits qui pourraient être exploités pour attaquer le système.

Isolation incorrecte Un autre vecteur d'attaque est la mise en place incorrecte des namespaces. Ainsi, si un conteneur est dans la même namespace PID que l'hôte, il est en mesure de voir les processus s'exécutant dans l'hôte (`ps`, ...) et d'interagir directement avec eux (`kill`, ...), allant ainsi à l'encontre des principes de la conteneurisation.

Secrets en clair La mise en place de secrets en clair peut permettre des vols de secrets par les entités autorisées à accéder à la configuration du conteneur. Ces entités sont alors en mesure d'utiliser ces données de manière malveillante, par exemple pour se connecter indûment à une base de données et y voler les données.

2.3.2 Vulnérabilités logicielles

Les environnements conteneurisés exécutent des applications logicielles qui comme dans tout environnement peuvent poser des problèmes de sécurité.

Vulnérabilité dans l'image utilisée Certaines images peuvent contenir des vulnérabilités non corrigées rendant le système attaquable. Ce risque est renforcé lorsque les images de conteneur sont téléchargées depuis des hubs logiciels comme DockerHub, où la sécurité de l'image n'est pas forcément (correctement) gérée, et peut donc contenir des vulnérabilités. Ainsi d'après [122], 67% des images "community" et 45% des images "official" contiennent au moins une vulnérabilité "haute" ou "critique". Cela est cohérent avec l'analyse de [109] datant de 2017 montrant que plus de 85% des images DockerHub étaient affectés par au moins une vulnérabilité de sévérité "haute" ou "critique".

Image non mise à jour Un autre enjeu de sécurité majeur est le maintien à jour des images. En effet, une image à jour expose son utilisateur à toutes les vulnérabilités trouvées depuis la dernière mise à jour. En environnement conteneur cette problématique est encore plus importante car les délais de mises à jour peuvent se cumuler. En effet, une fois une faille détectée, le développeur logiciel doit corriger le logiciel affecté. Il doit ensuite mettre à jour la version de l'image disponible sur le hub logiciel. L'utilisateur de l'image doit alors mettre à jour son image vers la dernière version et redémarrer ses conteneurs affectés pour utiliser la version corrigeant la vulnérabilité. Le cumul de ces délais fait que des images vulnérables peuvent parfois être utilisées en production pendant une longue période de temps. Selon [122], 47% des images "community" et 15% des images official de DockerHub n'ont pas été mises à jour depuis plus de 200 jours, laissant l'image vulnérable au vulnérabilités trouvées dans ce laps de temps.

Logiciels malveillants Certaines images ne provenant pas d'une source de confiance (cloud multi-tenant, hub d'images) peuvent contenir des logiciels malveillants pouvant tenter d'attaquer le système, avec les mêmes conséquences que dans un environnement classique.

Authentification insuffisante Les mécanismes d'authentification de certaines ressources conteneurisées peuvent être insuffisants, permettant à des attaquants de voler des données ou de réaliser des actions non souhaitées.

2.3.3 Failles système

En plus des vulnérabilités logicielles et liées à la conteneurisation, il est possible d'attaquer le moteur de conteneurisation (ou d'orchestration) lui-même via les biais suivants :

Vulnérabilité de l'hôte Si l'hôte contient une vulnérabilité, il est souvent possible de l'exploiter depuis un conteneur. Ceci est dû au fait que le conteneur utilise directement le noyau de l'hôte et donc peut exploiter les fonctionnalités de l'hôte comme dans un environnement classique.

Vulnérabilité dans le moteur de conteneurisation/orchestration Les vulnérabilités des moteurs de conteneurisation sont relativement rares mais existent. Celles-ci

peuvent notamment permettre de prendre le contrôle sur l'hôte parfois en tant qu'administrateur et impacter les conteneurs co-localisés.

Mauvaise gestion des droits du moteur de conteneurisation/orchestration Une utilisation arbitraire ou insuffisamment protégée par des utilisateurs qui ne sont pas de confiance de la ligne de commande du moteur de conteneurisation ou d'orchestrateur peut poser des problèmes de sécurité car elle permet de mettre en place des conteneurs très privilégiés pouvant trivialement compromettre le système. Par exemple, en laissant un attaquant créer un conteneur via `docker run -it --rm --privileged -v /:/host alpine` , le nouveau conteneur sera créé avec tous les fichiers de l'hôte montés dans `/host`. Il est possible de lire ou réécrire arbitrairement sur tous les fichiers de l'hôte depuis ce conteneur. Il est donc nécessaire de restreindre l'utilisation de la ligne de commande et des fonctionnalités similaires au seul administrateur.

2.4 Taxonomie des défenses conteneur

Dans cette section et les suivantes, nous nous intéressons aux différentes méthodes de défense des conteneurs. Nous proposons une nouvelle taxonomie pour formaliser les différentes méthodes de protection des conteneurs depuis l'hôte.

En plus des techniques de défenses fournies par l'hôte, il est possible de mettre en place des politiques de sécurité directement depuis l'intérieur du conteneur (durcissement logiciel par la désactivation des fonctionnalités superflues, vérification interne d'intégrité, ...). Toutefois, de telles défenses ne peuvent pas être considérées comme très robustes d'un point de vue sécurité puisque si le conteneur protégé devient compromis, il est possible de désactiver les protections mises en place, sans que cela ne soit détectable par l'infrastructure. Par ailleurs, l'hôte n'a pas la garantie que les politiques mises en place pour le conteneur sont réellement et correctement appliquées. De manière générale, l'hôte ne peut pas avoir confiance en ce qui est exécuté au-dessus de la frontière de virtualisation et donc dans les protections appliquées dans le conteneur. De fait, dans cette thèse, nous nous intéresserons uniquement aux défenses appliquées au niveau de l'hôte, excluant ainsi les défenses appliquées depuis le conteneur lui-même.

Puisque l'hôte et le conteneur communiquent à travers la frontière de virtualisation, située au niveau OS dans le cadre de la conteneurisation, la frontière de virtualisation est un point intéressant à considérer pour classifier le mécanisme de défense. Afin de

différencier ces différents mécanismes, nous nous posons la question suivante : "Quel type de données doivent traverser la frontière de virtualisation afin de mettre en place cette politique ?". Ceci fait ressortir les trois catégories suivantes :

— **Défense basée sur la configuration** Les politiques sont définies et mises en place au niveau de l'hôte grâce à un fichier de configuration ou une chaine de caractères. De telles politiques sont appliquées au lancement du conteneur.
— **Défense basée sur les règles** Le conteneur charge à travers la frontière de virtualisation des règles, généralement sous forme de chaines de caractères. Ces règles sont interprétées et appliquées par un code existant dans le noyau. Il est possible de rajouter de telles règles à tout moment.
— **Défense basée sur le code** Le conteneur charge à travers la frontière de virtualisation du code qui constitue le "cœur" des politiques de sécurité. Ces politiques sont appliquées quand le conteneur exécute des opérations noyau protégées par cette politique.

Le processus pour classifier un mécanisme de défense de niveau infrastructure selon cette taxonomie est illustré par la figure 2.1. Ainsi, si aucune donnée ne circule entre le conteneur et l'hôte à l'exécution du conteneur, le mécanisme de défense appartient forcément à la famille des configurations. Dans le cas contraire, l'élément de distinction entre les politiques est de savoir si les politiques sont exécutables (défense basée sur le code) ou juste interprétées en tant que règles (défense basée sur les règles). Dans les sections suivantes, nous présentons ces trois types de défense. A la lumière de cet état de l'art, nous présentons les avantages et les limitations de ces approches en section 2.8.

2.5 Défenses basées sur la configuration

Les défenses basées sur la configuration permettent à l'hôte de personnaliser les propriétés d'un conteneur à travers un fichier de configuration. Cette méthode est la plus universelle et est supportée par tous les moteurs de conteneurisation actuels. Il est possible de limiter largement la surface d'attaque d'un conteneur en le configurant c'est à dire en le mettant dans les bonnes namespaces, en appliquant une politique AppArmor restrictive, en lançant le conteneur en mode non-privilégié, ...

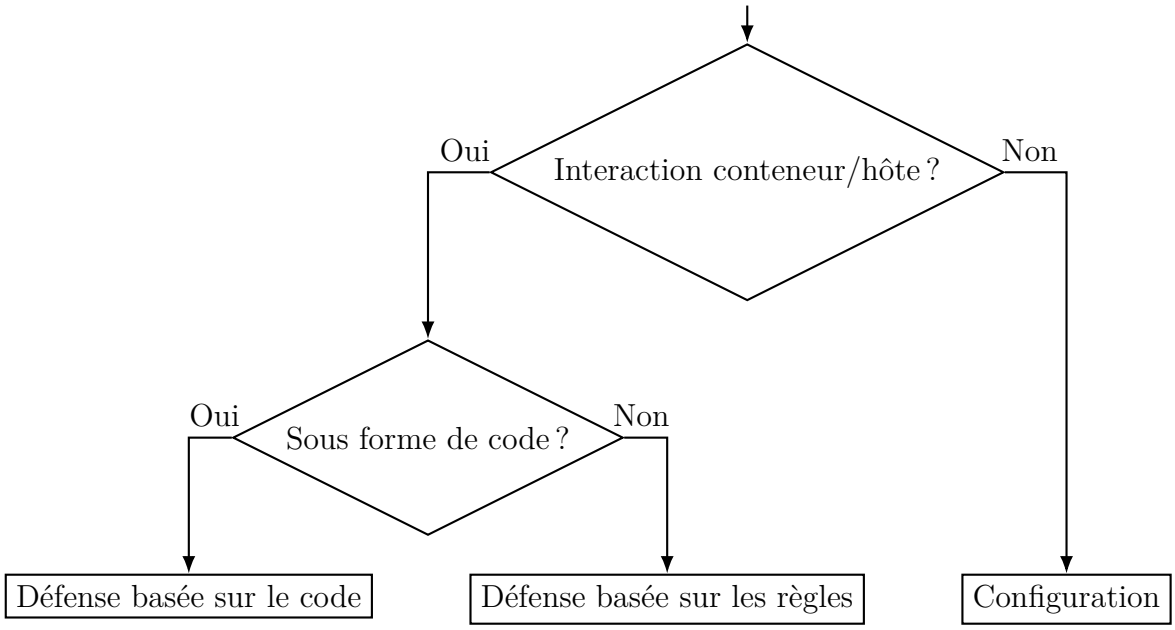


FIGURE 2.1 – Taxonomie des défenses de niveau infrastructure

2.5.1 Configuration du moteur de conteneurisation

Les moteurs de conteneurisation usuels permettent de mettre en place un grand nombre d'options de sécurité à travers une configuration du conteneur. Par exemple, Docker permet à travers un fichier DockerFile ou des arguments à la ligne de commande, de personnaliser par conteneur toutes les abstractions à la base de la conteneurisation. Il est donc possible de personnaliser les namespaces d'un conteneur, ses capabilités, ses cGroups et ses politiques de sécurité pour les Linux Security Modules existants. Les orchestrateurs de conteneurs permettent aussi de configurer précisément les conteneurs à lancer. Par exemple, Kubernetes permet d'appliquer des configurations à des ensembles de conteneurs à travers le 'docker compose'.

Les configurations mises en place pour les conteneurs sont appliquées pour l'intégralité du cycle de vie des conteneurs et ne peuvent donc pas être modifiées. Cela est lié à la philosophie des conteneurs qui implique d'instancier un conteneur pour une configuration donnée et donc de redémarrer le conteneur en cas de besoin de modification de sa configuration.

2.5.2 Vérification d'intégrité

Afin de garantir qu'aucune modification malveillante n'a été faite sur des conteneurs, l'intégrité des fichiers des conteneurs est une propriété à vérifier. Sur des systèmes classiques, la vérification d'intégrité se fait avec des modules comme le LSM IMA (Integrity Measurement Architecture) [100]. IMA permet de notamment mesurer le hash des fichiers, potentiellement à l'aide d'un TPM, et de comparer ces valeurs avec des "bonnes" valeurs. IMA permet donc de détecter toute modification non prévue. Toutefois, IMA est conçu pour protéger tout le système et n'est pas disponible pour protéger des conteneurs individuels.

DIVE [29] est un framework permettant à un orchestrateur de conteneurs de vérifier l'intégrité des fichiers d'un conteneur donné sur un nœud de calcul donné. DIVE s'appuie sur une version modifiée d'IMA supportant les conteneurs et un framework d'attestation (OpenAttestation). L'intégrité des conteneurs est vérifiée au niveau de l'hôte, c'est à dire sans aucune interaction avec le conteneur. Ainsi, si un conteneur est détecté comme compromis, il est possible de demander à l'orchestrateur de reconstruire uniquement ce conteneur (et non pas le système complet). Ainsi, grâce à DIVE, l'infrastructure est en mesure de vérifier périodiquement que les conteneurs ne sont pas compromis. Toutefois, puisque DIVE ne surveille que l'intégrité des fichiers, il n'est pas en mesure de détecter des modifications qui ne seraient faites qu'en mémoire, ce qui représente une part significative des attaques (ex. buffer overflow, injection de code, ...). Ainsi, DIVE ne constitue pas une solution aboutie pour vérifier les charges applicatives des conteneurs.

D'autres approches permettent de vérifier l'intégrité de conteneurs. Ainsi CloudVaults [65] permet de réaliser de la vérification d'intégrité des fichiers des conteneurs et évite à l'aide de Service Graphs Chains que les conteneurs puissent récupérer les informations des mesures d'intégrité des autres conteneurs, qui peuvent sinon être utilisées par des attaquants pour inférer des informations sur les autres conteneurs, brisant ainsi l'isolation de ces primitives. Plus récemment, IPE [74] permet aussi de monitorer l'intégrité de conteneurs. Contrairement à d'autres frameworks comme IMA, IPE n'a pas de dépendance avec les metadonnées du système de fichiers et réutilise avantageusement les fonctionnalités du noyau pour gérer les signatures, réduisant sa surface d'attaque en comparaison d'IMA.

2.5.3 Sécurité Réseau

Plusieurs frameworks tentent d'améliorer la sécurité réseau des conteneurs. Parmi ceux-ci, on peut citer :

Cilium [21] est un framework de sécurité réseau qui permet de mettre un place un proxy de niveau 7 utilisable pour confiner de manière fine les conteneurs. Il est notamment possible de configurer pour chaque conteneur quels points d'accès d'une API REST sont disponibles. Cilium permet donc de limiter la surface d'attaques au minimum en ne gardant que les fonctionnalités réellement utilisées par le conteneur. Une telle protection permet également d'éviter qu'un accès malicieux à une API REST mal conçue puisse permettre une attaque, notamment de type Confused Deputy [50], ce qui est utile en environnement microservices. Puisque le filtrage des paquets réseau permettant de réaliser ce proxy est basé sur eBPF (extended Berkeley Packet Filter, présenté en section 2.9.3) et peut fonctionner avec XDP (eXpress Data Path, une version d'eBPF optimisée pour le filtrage rapide de paquets réseau), Cilium dispose de très bonnes propriétés de performances, en comparaison des approches traditionnelles comme iptables. Les logs conservés par Cilium sur l'activité réseau des conteneurs peuvent par ailleurs être utiles pour réaliser des analyses forensiques et des audits de sécurité. Malgré tout, le champ d'action de Cilium reste très centré sur le filtrage réseau. Ce framework doit donc être associé à d'autres pour améliorer de manière significative la sécurité des conteneurs.

BASTION [76] est un autre framework permettant d'instancier une pile réseau par conteneur, isolée de la pile réseau de l'hôte. BASTION permet également de définir des politiques de sécurité réseau à grain fin par conteneur (et non par ip). Il est possible de configurer les ressources réseau distantes auxquelles le conteneur peut accéder. Les dépendances entre les conteneurs peuvent être récupérées automatiquement du moteur de conteneurisation pour mettre en place automatiquement les règles de sécurité. Enfin, ce framework vérifie que les paquets viennent bien des conteneurs, évitant les paquets forgés (spoofing). BASTION reste aussi très spécifique au réseau et doit être associé à des protections système pour améliorer la sécurité conteneurs de manière significative.

2.6 Défenses basées sur le code

Les approches basées sur le code permettent aux conteneurs de charger du code vers le noyau. Quand le conteneur exécute une opération donnée sur un objet donné, le code chargé par le conteneur est alors exécuté pour vérifier certaines propriétés de sécurité

et donc la légitimité de l'opération. Ainsi, cette technique permet à un conteneur de faire appliquer de manière flexible sa propre logique de sécurité. Cette technique est notamment prometteuse en environnement multi-tenant où les besoins de sécurité des différents conteneurs peuvent être très différents.

Puisqu'une telle approche permet à un conteneur donné de mettre en place sa propre politique de sécurité via du code, il est possible de limiter la surface d'attaque dudit conteneur ou le protéger contre des attaques spécifiques sans nécessiter de dépendances à d'autres frameworks de sécurité. Il est donc possible d'appliquer ces politiques de sécurité programmables sur toutes les machines (et pas seulement les machines où les bons modules sont installés et activés). Par ailleurs, puisque les approches basées sur le code ne nécessitent pas d'interpréter de règles et peuvent être directement exécutées, il est possible d'obtenir de meilleures performances avec ce type d'approche.

La technique la plus connue de mise en place de code, eBPF [4], est une technique standard du noyau Linux et est aujourd'hui considérée comme relativement mature. Le design de ce langage est présenté en section 2.9.3. Toutefois, si les codes exécutés et notamment eBPF sont des codes limités qui ne sont *en théorie* pas en mesure d'attaquer le noyau, cette technique permet à une entité qui n'est pas de confiance d'exécuter des codes directement dans le kernel. L'exploitation d'une vulnérabilité dans le vérifieur eBPF (par exemple la CVE-2021-3490 [85]) pourrait donc avoir des conséquences très importantes. Par ailleurs, les limitations d'eBPF restreignent un peu sa flexibilité. Finalement, écrire un code eBPF correct est plus complexe qu'écrire une règle de sécurité, même si des helpers peuvent parfois faciliter la génération de code.

2.6.1 Landlock LSM version<14

Landlock LSM [103] est un framework logiciel dont l'objectif est de permettre à tous les processus, y compris non privilégiés de s'auto-cloisonner afin de limiter utilement leurs droits en cas de compromission.

Dans les versions initiales du framework, ce comportement était implémenté en permettant le chargement de programmes eBPF par hiérarchie de processus dans le noyau du système. Ces programmes eBPF étaient attachés à des événements LSMs. Landlock LSM a originellement été proposé en 2018 et a été intégré dans le noyau avec la version 5.13 [27] dans un mode plus restreint de filtrage par règles. En effet, une partie de la communauté Linux s'oppose à la mise en place d'eBPF non privilégié pour des raisons de sécurité. Pour certains, eBPF n'est pas assez mature pour pouvoir être exécuté sans

privilèges dans le noyau, pour d'autres eBPF ne le sera même *jamais* [26], Landlock a donc décidé d'abandonner l'utilisation de ce langage en faveur d'un système de "règles" plus classique. Si l'abandon d'eBPF est un choix qui peut se justifier d'un point de vue technique et qui a probablement permis l'intégration de Landlock LSM dans le noyau Linux, il rend toutefois son architecture moins unique. Nous faisons donc le choix de présenter dans cette section le mode initial de défense basée sur le code, non retenu par Linux mais qui ouvre la voie à une sécurité noyau programmable.

Puisque Landlock fonctionne au niveau LSM et non pas au niveau appel système, les programmes Landlock ne sont exécutés que si l'appel système réalise une opération valide. Ainsi, si l'appel système aboutit sur une erreur, le hook LSM n'est pas déclenché permettant de n'appliquer des politiques que des appels valides. L'utilisation d'une architecture LSM telles qu'utilisée par Landlock permet d'accéder à l'objet accédé et aux structures noyau pour faire des vérifications de politiques de sécurité, au contraire des filtrages classiques par appel système tel que réalisés par seccomp-bpf où le filtrage ne peut pas déréférencer les pointeur noyau et ne peut donc pas faire un filtrage à grain fin. L'approche Landlock est donc très intéressante au sens où chaque processus est en mesure durant tout son cycle de vie d'empiler des politiques de sécurité à grain très fin non désactivables. Ce framework permet donc en théorie de faire du contrôle d'accès beaucoup plus fin que l'existant pour améliorer la sécurité des systèmes.

Une fois les politiques mises en place, leur vérification est réalisée comme suit : lors d'un appel système (ici `open()`), juste avant l'accès au fichier, le hook LSM `file_open` est exécuté. Toutes les politiques eBPF liées à ce hook sont alors exécutées. L'accès est accordé au programme uniquement si tous les programmes eBPF l'acceptent.

Nos contributions ont été inspirées par l'approche Landlock et comportent des similarités avec ce framework. Toutefois, Landlock possède des limitations levées par SNAPPY.

— Par design, Landlock utilise le langage eBPF pour implémenter ses politiques de sécurité. Puisque pour des raisons de sécurité ce langage est très limité, les politiques exprimables par Landlock sont elles aussi limitées.
— Puisque Landlock permet de mettre en place dans le noyau des programmes eBPF *arbitraires*, la surface d'attaque créée par ce framework est importante, ce qui a justifié un abandon d'eBPF en tant que moteur de politiques. Notre contribution, SecuHub, présenté dans cette thèse, limite la distribution de politiques de sécurité à une unique entité de confiance, où les politiques peuvent être aisément auditées, évitant ainsi ce problème.

— Seuls quelques hooks LSM sont prévus liés notamment à la gestion des fichiers, du ptracing, ... Ceci limite donc la versatilité du framework Landlock.
— Le filtrage par hiérarchie de processus peut poser des problèmes si des processus sont créés à travers des démons système comme c'est le cas avec certains moteurs de conteneurisation dont Docker. En effet, des processus créés à partir de tels démons système peuvent ne plus avoir de lien de parenté avec le processus ayant demandé leur création. Si des politiques de sécurité étaient mises en place pour le processus parent, il est alors possible que le nouveau processus n'ait plus ces politiques mises en place, résultant en une élévation de privilège.

2.6.2 Mécanismes de défenses globaux eBPF

Plusieurs mécanismes de sécurité permettent d'appliquer des politiques de sécurité eBPF. Toutefois, celles-ci s'appliquent à l'intégralité du système et pas à un conteneur en particulier limitant largement leur intérêt pour la sécurité des conteneurs.

Il est possible de charger dans le noyau des programmes eBPF à de très nombreux points du noyau [57, 17]. Il est tout d'abord possible de mettre en place des programmes eBPF au niveau des appels systèmes afin de tracer leurs comportements. Il est par ailleurs possible de mettre en place des programmes eBPF pour tracer des fonctions, au début de leur exécution (via `fentry`) ou à la fin de leur exécution (via `fexit`). Depuis la version 5.7 du noyau (mai 2020), il est également possible de mettre en place des politiques de sécurité eBPF au niveau des hooks LSM [61]. Ces manières de mettre en place des programmes eBPF possèdent toutefois les limitations suivantes : leur utilisation requiert d'être un utilisateur privilégié pour des raisons de sécurité et elles s'appliquent à tout le système, ce qui signifie qu'il n'est pas possible de mettre en place des règles de sécurité pour protéger des conteneurs spécifiques.

KRSI [110] permet à l'administrateur d'appliquer des politiques eBPF aux hooks LSM afin de détecter les symptômes d'une attaque potentielle, par exemple un processus utilisant `LD_PRELOAD`. Toutefois, de telles politiques doivent être appliquées à l'intégralité du système et non à un simple conteneur. De plus, KRSI ne peut pas être utilisé par une entité non privilégiée.

2.6.3 BPFBox et BPFContain

Bpfbox [36] est un framework permettant de définir des politiques basées sur eBPF visant à confiner des processus individuels. Les politiques de bpfbox peuvent être appliquées au niveau LSM, appel système, fonction noyau ou fonction en espace utilisateur. L'idée de bpfbox est donc de pouvoir charger des politiques eBPF à tout endroit du noyau eBPF. Contrairement à Landlock LSM, les politiques LSM peuvent uniquement être utilisées par des processus privilégiés, empêchant leur utilisation par des conteneurs non privilégiés. BPFContain [35] explore un cas d'usage avec le framework bpfbox : le confinement des conteneurs, permettant de restreindre les comportements autorisés par ces derniers (et non pas de processus individuels comme avec Bpfbox). Toutefois, puisque BPFContain n'est utilisable que par un utilisateur privilégié, les conteneurs eux-mêmes ne peuvent pas s'en servir pour limiter leur surface d'attaque.

Bpfbox et BPFContain permettent de définir les politiques de sécurité dans un langage intermédiaire afin de définir plus simplement des politiques eBPF. Si cette couche d'abstraction peut effectivement simplifier l'écriture de politiques de sécurité en la rapprochant d'un système de règles, elle limite la flexibilité des politiques pouvant être écrites de par la faible expressivité de ce langage.

2.6.4 Seccomp-bpf

Seccomp-bpf [46] est un mécanisme de sandboxing intégré au noyau Linux permettant de limiter la surface d'attaque des processus/conteneurs en bloquant des appels systèmes à l'aide de code eBPF. Toutefois, nous avons déjà expliqué en section 1.5.4 pourquoi le filtrage par appel système était une approche moins flexible que le filtrage par module de sécurité Linux. Par ailleurs, pour des raisons de sécurité ces politiques ne peuvent pas déréférencer les pointeurs kernel. Par exemple, le deuxième argument de l'appel système `open()` contient l'adresse du chemin du fichier à ouvrir. De fait, les politiques de sécurité Seccomp-bpf ne permettent pas de limiter finement les accès d'un conteneur.

2.6.5 Falco

Falco [101] est un traceur d'appels système visant à détecter les menaces logicielles. Les politiques appliquées peuvent être basées sur un module noyau ou sur du code eBPF. Il est possible de mettre en place des règles de sécurité venant d'une bibliothèque de règles fournies par la communauté ou par sysdig. Falco vise à surveiller le comportement des

conteneurs (commandes exécutées, connections réseau, ...) afin de détecter d'éventuelles attaques. Falco est en mesure de déclencher des alertes en cas d'activité suspicieuse visant à faciliter la réalisation d'audit de sécurité.

Malgré tout, tout comme Seccomp-bpf, Falco hérite des limitations liées au filtrage par appel système. De plus, les alertes remontées par Falco sont seulement détectées, mais pas corrigées. Par ailleurs, les politiques de Falco ne sont pas à états, limitant les attaques pouvant être détectées par un tel mécanisme.

2.7 Défenses basées sur des règles de sécurité

Les défenses basées sur des règles de sécurité permettent aux conteneurs d'appliquer des politiques de sécurité sous forme de chaines de caractères qui sont interprétées et exécutées par le noyau hôte. Les règles peuvent être écrites selon un formalisme précis, par exemple `{file:"/etc/shadow", expected:"0x1C3A9F1E"}` permet d'indiquer le hash attendu du fichier `/etc/shadow` . De telles règles sont généralement plus simples à écrire et à auditer que des politiques programmables. Les administrateurs ont la possibilité de gérer les frameworks mis à disposition et donc le type de règles qui peuvent être mises en place pour un scénario d'utilisation donné. Puisque l'hôte met en place de telles politiques proactivement, il est en mesure de vérifier qu'elles sont correctes.

Toutefois, les approches basées sur les règles sont moins flexibles que celles basées sur le code puisqu'il n'est pas possible d'exprimer une aussi grande diversité de politiques avec un ensemble de règles qu'avec un langage complet tel qu'eBPF. Puisque les règles sont mises en place dans le kernel, il n'est par ailleurs pas possible d'ajouter des types de règles de sécurité sans que l'administrateur ne doive modifier et recompiler le kernel. Une telle approche est donc moins évolutive qu'une approche basée sur le code, chargeable dynamiquement et des conteneurs pourraient ne pas avoir à disposition dans le kernel les bons outils basés sur les règles nécessaires à leur sécurité.

2.7.1 Security Namespaces

Security Namaspace [115] est un framework visant à 'namespacer' les modules LSM, permettant notamment de les rendre disponibles aux conteneurs. Ceci est réalisé en créant une nouvelle namespace orientée sécurité `security_ns` et en tentant d'adapter manuellement les LSMs existants à cette namespace. Puisque la namespace de sécurité mise en

place est une namespace *réelle* (et non émulée), il est possible de l'utiliser comme une namespace classique, par exemple la création d'une namespace s'effectue avec un flag sur les appels systèmes `clone()`, `fork()` ou `unshare()`. Il est donc possible de mettre en place des politiques pour protéger spécifiquement des namespaces. Lors de la création d'une nouvelle namespace, la nouvelle namespace hérite des politiques de la namespace parent, évitant ainsi toute possibilité d'élévation de privilèges via ce mécanisme. Il est à noter que Security Namespace ne fournit pas de modèle général pour l'adaptation des LSMs aux conteneurs et un tel modèle serait complexe à implémenter puisque les différents LSMs fonctionnent de manière différente et ont besoin d'utiliser et de stocker dans la namespace des données très différentes. Des prototypes *spécifiques* à certains modules LSMs (AppArmor et IMA) ont été implémentés.

Security Namespaces permet de mettre en place des politiques de sécurité pour protéger des namespaces données en chargeant ces politiques sous forme de chaines de caractères via des interfaces dédiées. A l'utilisation, lorsqu'une opération LSM sensible est réalisée, par exemple une ouverture de fichier, un composant nommé proxy manager récupère la liste des namespaces ayant la visibilité sur l'objet, c'est à dire ici, le fichier ouvert. L'intégralité des politiques liées à cet objet est alors exécutée. L'accès à l'objet n'est accordé que si toutes les namespaces valident cet accès.

Une fonctionnalité clé de Security Namespaces est la détection automatique d'incompatibilités de politiques entre elles pour à un fichier particulier, par exemple read-only vs read-write, qui pourraient engendrer un faux sentiment de sécurité ou un Déni de Service. Security Namespace permet aussi, via un système d'autorité de déclarer la propriété d'une namespace sur un objet, permettant de rendre les règles de sa namespace obligatoires (mandatory), c'est à dire applicables également aux autres namespaces. Ce mécanisme permet également de mettre en place des politiques de sécurité pour les autres namespaces. L'utilisation du système d'autorité nécessite la possession d'une capabilité système, pour éviter qu'un processus ne puisse créer un déni de service sur le système via ce mécanisme. Nous pensons toutefois que ce mécanisme reste dangereux car il est parfois légitime pour une namespace de devoir déclarer l'autorité sur un objet, rendant nécessaire l'obtention de cette capabilité. Avec un tel mécanisme, cette capacité peut alors facilement être abusée pour rendre le système inutilisable. Pour cette raison, ainsi que pour des raisons de performances, notre contribution SNAPPY n'utilise pas un tel mécanisme d'autorité.

Pour qu'un module LSM devienne compatible avec le framework security namespaces,

ce module est modifié pour que les données soient récupérées depuis cette namespace. Il existe toutefois des difficultés faisant qu'il n'est pas possible d'adapter tous les modules LSM à Security Namespace. Par exemple, les Modules de Sécurité Linux majeurs stockent leurs politiques de sécurité dans le champ `security` des structures de données qu'elles protègent. Puisque ce champ est un pointeur unique, il n'est pas possible de faire cohabiter plusieurs modules sans modifications additionnelles et une namespace ne résout pas ce problème. Similairement, des variables globales réseau comme `secids` ou `secmarks` rendent cet empilage impossible sans modifications architecturales majeures.

Il est à noter que de manière contemporaine à Security Namespaces, l'adaptation aux conteneurs a notamment été réalisée pour AppArmor dans la mainline Linux. Cette namespacisation est réalisée à l'aide d'une émulation de namespace. L'utilisation d'une vraie namespace, comme dans Security Namespaces garde toutefois des avantages significatifs. Cela réduit la surface d'attaque en réutilisant une primitive connue, permet un code plus simple et clair, peut améliorer les performances en ne nécessitant pas la gestion de ces namesapaces émulées [38].

En conclusion, Seurity Namespace est une approche intéressante qui montre notamment en quoi une namespace de sécurité pourrait être utilisée pour améliorer la sécurité des hiérarchies de processus et des conteneurs et nous pensons que le principe d'une namespace devrait donc être adoptée dans la mainline Linux. Toutefois, puisque ce framework nécessite des modifications noyau importantes dans les modules à adapter et fonctionne uniquement avec certains modules LSM, c'est à dire que l'approche Security Namespace n'est pas générique, il ne peut pas être vu comme une solution générale à la sécurité des conteneurs.

2.7.2 Landock LSM version≥14

Les versions récentes de Landlock [102] depuis la version 14 ont abandonné le moteur eBPF pour utiliser un système de règles.

Avec Landlock, il est possible de mettre en place des règles grâce à 3 nouveaux appels systèmes : `landlock_create_ruleset()`, `landlock_add_rule()` et `landlock_restrict_self()`. Ce nouveau système permet d'appliquer des ensembles de règles (ruleset) appliqués au processus courant et à ses descendants. Ainsi, lors de l'application de règles, les objets sur lesquels la règle s'applique sont "taggés". Les accès futurs peuvent alors être contrôlés en vérifiant la valeur de tels tags sur la hiérarchie de l'objet accédé et en "additionnant" les tags.

Dans la version 5.13 de landlock, la première intégrée dans le noyau Linux, il est possible de mettre en place des règles pour restreindre l'accès à un répertoire et les répertoires sous-jacents [27] du processus courant et de ses descendants. D'autres cas d'usages devraient être mis en place lorsque le framework gagnera en maturité.

D'un point de vue de l'espace utilisateur, Landlock fournit des bibliothèques permettant de charger des règles Landlock dans le noyau via des programmes Go et Rust, simplifiant l'application de règles.

Si ces dernières itérations du module Landlock ont permis son acceptabilité dans le noyau, tout en conservant son objectif initial de permettre du sandboxing *non privilégié*, elles rendent aussi ce framework moins unique. Dans cette thèse, nous prendrons donc plus particulièrement en référence les versions anciennes de LandLock LSM basées sur eBPF, décrites en section 2.6.1.

2.7.3 Pledge

Pledge [30] est un framework de sécurité pour OpenBSD fournissant un mécanisme simple de sandboxing pour restreindre les comportements autorisés des processus individuels, limitant ainsi les risques d'élévation de privilèges et la surface d'attaque. Contrairement à seccomp-bpf [46] présenté en 2.6.4, pledge se base sur l'autorisation d'un ensemble de "comportements" et pas celle d'appels systèmes. Par exemple, la règle `stdio` permet l'accès à libc, `rpath` l'accès en lecture seule à des fichiers, ... Il est possible de mettre en place ces politiques à l'aide de l'appel système dédié `pledge()`. Il est par ailleurs possible d'appeler plusieurs fois pledge afin de réduire encore les capabilités d'un processus, permettant par exemple d'interdire pendant la boucle principale des comportements qui ne seraient légitimes qu'à l'initialisation.

À l'exécution, lorsqu'un appel système est réalisé, un hook est en mesure de vérifier la conformité de l'accès vis-à-vis des politiques. Toutefois, pledge ne contrôle que des processus individuels et non des hiérarchies de processus ou des conteneurs. De fait, il est possible de contourner les politiques pledge via la création d'un nouveau processus, puisque ce nouveau processus n'hérite pas des politiques du processus parent. Les comportements pouvant être filtrés avec Pledge restent assez gros-grain. Il n'est donc pas possible de faire du filtrage très précis avec ce mécanisme. Enfin, Pledge reste spécifique à OpenBSD. Pledge ne constitue donc pas non plus une solution parfaite pour limiter la surface d'attaque du processus.

2.8 Comparaison des différentes approches et frameworks

Dans cette section, nous comparons les différentes approches (section 2.8.1) et les différents frameworks (section 2.8.2) précédemment présentés dans cet état de l'art.

2.8.1 Comparaison des approches existantes

Dans ce chapitre, nous avons présenté l'état de l'art des mécanismes de défense applicables aux conteneurs et catégorisé les différents frameworks de l'état de l'art selon le type de politiques : les configurations, les politiques basées sur des règles et les politiques basées sur du code. Ces trois approches présentent des avantages et des inconvénients récapitulés dans le tableau 2.2.

	Configurations	Règles	Code
Simplicité	●	◐	○
Granularité	○	◐	●
Maturité	●	◐	○
Performances	●	○	●

Table 2.2 – Comparaison des différentes approches de sécurité

Puisque les configurations peuvent être utilisées directement dans les moteurs de configuration, leur utilisation est très simple. Les approches basées sur les règles, sont légèrement plus complexes puisqu'elles nécessitent de mettre en place des règles manuellement dans un format précis. Les approches basées sur le code sont les plus complexes et nécessitent le développement d'un code réalisant la fonction de sécurité. Pour cette raison, l'état de la technique est aujourd'hui plus avancé pour la configuration et les approches basées sur les règles que celui des approches basées sur le code. En termes de granularité, les approches basées sur le code sont les plus puissantes de par la versatilité du code. Enfin, en termes de performances, les techniques basées sur les règles sont généralement moins performantes que les autres puisqu'elles impliquent que les règles soient interprétées et pas directement exécutées, ce qui rajoute un niveau d'abstraction qui impacte les performances.

En tout état de cause, il convient de comprendre que ces différentes approches ne sont pas du tout incompatibles et afin d'atteindre un niveau élevé de sécurité, il est souhaitable

d'utiliser plusieurs approches simultanément.

Dans la suite de cette thèse, nous travaillerons plus profondément sur les approches basées sur le code, car elles permettent de mettre en place des politiques très fines et constituent une piste de recherche prometteuse [49]. Les frameworks SNAPPY et SecuHub permettent de mettre en place du code avec des avantages sur les approches existantes.

2.8.2 Comparaison des frameworks existants

	Basé sur : Config(C), Règles(R) ou Code(X)	Champ d'action	Extensibilité	Adaptation aux conteneurs	Utilisation interactive non privilégiée	Maturité
Outils de configuration	C	●	○	●	○	●
DIVE [29]	C	○	○	●	○	○
CloudVaults [65]	C	○	○	●	○	○
IPE [74]	C	○	○	○	○	○
Cilium [21]	C	◐	○	●	○	●
Security Namespaces [115]	R	◐	○	●	○	○
BASTION [76]	R	○	○	●	○	○
Landlock v>14 [102]	R	◐	○	●	●	●
Landlock v<14 [103]	X	●	◐	●	●	○
eBPF noyau [61]	X	●	◐	○	○	●
KRSI [110]	X	●	◐	○	○	●
BPFBox + BPFContain [36, 35]	X	●	◐	●	○	○
SeccompBPF [46]	X	●	○/◐	●	●	●
Falco [101]	X	●	◐	●	○	●

Table 2.3 – Comparaison des différents frameworks de sécurité

La table 2.3 compare les avantages et limitations des différents frameworks présentés dans ce chapitre. La première colonne de cette table indique le **type de l'approche** présentée selon la taxonomie définie en 2.4. La deuxième colonne indique le **champ d'action** des politiques. Les frameworks permettant de mettre en place des politiques sur tous les appels systèmes ou les hooks LSM sont considérés comme ayant un champ d'action sa-

tisfaisant. C'est aussi le cas des configurations de par la diversité des approches pouvant être mises en place. Puisque Cilium et les versions récentes de Landlock ne permettent de mettre en place des politiques que pour des champs d'applications précis, nous considérons leur champ d'application comme moyen. Nous considérons les frameworks ne permettant qu'un contrôle particulier (intégrité) comme ayant un champ d'action restreint. La troisième colonne indique **l'extensibilité** des frameworks, c'est à dire dans quelle mesure il est possible de rajouter des fonctionnalités au framework sans modifier le logiciel ou le noyau. Aucune approche de l'état de l'art n'a une extensibilité très forte car les approches programmables présentées ici héritent des limitations d'eBPF liées à la sécurité, leur extensibilité reste donc moyenne. Les approches basées sur des règles ou une configuration ont une extensibilité faible. Similairement, nous considérons que seccomp-bpf a une extensibilité faible puisqu'il ne permet pas de déréférencer les pointeurs kernel et ne permet donc pas d'exprimer des politiques à grain fin. La quatrième colonne indique si les approches sont **adaptées aux conteneurs** ou s'appliquent forcément à tout le système. La cinquième colonne indique s'il est possible d'utiliser l'approche depuis l'intérieur d'un conteneur non privilégié. Seuls Landlock et Seccomp-bpf valident ce critère. Finalement la dernière colonne indique si les frameworks sont matures pour une utilisation en production.

On constate que tous les frameworks présentés ici présentent des avantages et des limitations et aucun n'est parfait. Nous cherchons avec le framework SNAPPY à améliorer cet état de l'art.

2.9 Programmabilité du noyau et opportunités de sécurité

2.9.1 Considérations générales

Dans de nombreux domaines de l'informatique, les fonctionnalités étaient originalement statiques et sont devenues avec le temps de plus en plus configurables puis programmables. Par exemple, dans le cadre du développement web,les sites étaient statiques avec très peu de fonctionnalités avant les années 2000 pour atteindre, notamment grâce à JavaScript des systèmes très dynamiques et riches en fonctionnalités que nous connaissons aujourd'hui [49].

La sécurité mise en place par le noyau semble aujourd'hui prendre un chemin similaire.

Les abstractions de sécurité liées au noyau initialement statiques (contrôle d'accès discrétionnaire, séparation mémoire, ...) sont devenues configurables de plus en plus finement (capabilities, namespaces). De nos jours, on assiste au développement de la sécurité programmable. Ce phénomène a été engendré par les modules de sécurité Linux et renforcé par eBPF. Aujourd'hui, le langage eBPF est utilisé dans de plus en plus de domaines et suscite un engouement. Nous présentons dans les prochaines sections en quoi les LSMs et eBPF peuvent améliorer la sécurité du système, illustrant ainsi pourquoi augmenter la programmabilité du noyau via des approches basées sur le code constitue un enjeu de sécurité important [49].

2.9.2 Linux Security Modules

Les Modules de Securité Linux, présentés en section 1.5.4, permettent de mettre en place des contrôles de sécurité personnalisés via des modules de sécurité programmables. De par l'architecture des LSMs, il est possible de rajouter des fonctionnalités de sécurité spécifiques à l'aide de code répondant aux hooks LSM, déclenchés lors d'événements système spécifiques. Puisque les LSM peuvent être empilés (sauf LSM majeurs), il est possible pour l'utilisateur de choisir le type de contrôle à mettre en place en utilisant un LSM ou un autre.

Les LSM ajoutent ainsi un premier niveau de programmabilité au noyau en permettant à l'utilisateur de choisir les fonctionnalités de sécurité qui lui conviennent pour son cas d'usage. Les LSM sont aujourd'hui très largement utilisés et permettent d'améliorer significativement la sécurité du système notamment en restreignant les comportements autorisés via des politiques de Mandatory Access Control.

Toutefois, il n'est pas possible de mettre en place un code de protection personnalisé sans devoir écrire un nouveau LSM sous forme de code noyau, ce qui est très complexe et nécessite de recompiler le noyau. Cela signifie qu'il n'est pas possible sur un boot donné de rajouter des fonctionnalités à l'architecture LSM.

2.9.3 extended Berkelet Packet Filter (eBPF)

2.9.3.1 Présentation technique

Le langage BPF (Berkeley Packet Filter) [72] en tant que binaire injectable dans le noyay est relativement ancien (1992). Ce langage est conçu pour ne pas être en mesure de compromettre le système. Cette propriété de sécurité s'acquiert au prix de limitations

importantes en termes de fonctionnalités. En outre, le langage BPF ne permet pas de faire d'arithmétique de pointeur, d'accéder à des structures noyau ou de faire des boucles (potentiellement) infinies.

Ainsi, à la compilation, un vérifieur est responsable de garantir toutes les propriétés de sécurité du code à charger. Si l'une de ces propriétés est violée, le code ne peut pas être chargé dans le noyau. De fait, un programme BPF n'est donc en théorie pas en mesure de compromettre le système.

Il est possible d'écrire des programmes BPF dans un sous-ensemble du C, simplifiant ainsi l'écriture de tels programmes. Ces programmes peuvent alors être compilés vers le code BPF à l'aide d'une chaine de compilation telle que `bcc` (BPF Compiler Collection) [57]. Un exemple de programme basique C et le code objet associé sont présentés respectivement dans le code 2.1 et 2.2. Ces codes montrent qu'il est possible d'écrire du code eBPF simplement. Le code objet obtenu reste compréhensible et peut être compilé à la volée, induisant de très bonnes performances.

```
SEC("lsmpp")
int my_main(int arg) {
    if(arg > 0)
        return 0;
    else if(arg < -3)
        return 1;
    return -1;
}
```

Code 2.1 – Exemple de programme eBPF C

```
llvm-objdump-12 -S  prog.o

0000000000000000 <my_main>:
; int my_main(int arg) {
    0:  67 01 00 00 20 00 00 00     r1 <<= 32
    1:  c7 01 00 00 20 00 00 00     r1 s>>= 32
    ;   if(arg > 0)
    2:  65 01 05 00 00 00 00 00     if r1 s> 0 goto +5 <LBB0_4>
    3:  b7 00 00 00 01 00 00 00     r0 = 1
    4:  b7 02 00 00 fd ff ff ff     r2 = -3
    ;   else if(arg < -3)
    5:  6d 12 01 00 00 00 00 00     if r2 s> r1 goto +1 <LBB0_3>
    6:  b7 00 00 00 ff ff ff ff     r0 = -1
```

```
0000000000000038 <LBB0_3>:
;    else if(arg < -3)
7:  95 00 00 00 00 00 00 00      exit

0000000000000040 <LBB0_4>:
; }
8:  b7 00 00 00 00 00 00 00      r0 = 0
9:  95 00 00 00 00 00 00 00      exit
```

Code 2.2 – Bytecode eBPF généré

Afin de pallier les limitations de BPF et permettre des comportements plus complexes bien que spécialisés, une amélioration à BPF nommée eBPF (extended Berkeley Packet Filter) [4] a été proposée en 2014. Celle-ci rajoute la fonctionnalité de helper eBPF [31]. Ces helpers consistent en un certain nombre de fonctions visant à réaliser un traitement très spécifique, par exemple récupérer le PID courant, écrire dans une map eBPF [28], ... Ces fonctions sont écrites dans le noyau et permettent d'améliorer la versatilité d'eBPF. Les helpers visent à gérer les cas d'usages courants d'eBPF pour en améliorer les fonctionnalités sans réduire la sécurité du langage. Toutefois, puisque ces helpers ne permettent de gérer un nombre limité de traitements spécifiques et qu'il n'est pas possible de rajouter d'helpers eBPF sans avoir à modifier et recompiler le noyau avec un nouvel helper, le langage eBPF reste très spécifique.

Il existe des versions d'eBPF spécialisées pour des cas d'usages particuliers. Ainsi, XDP [54] est une version d'eBPF spécialisée pour le filtrage des paquets réseau. Puisque le filtrage XDP est exécuté totalement en espace noyau, il permet du filtrage réseau sans changement de contexte et donc d'obtenir des performances supérieures aux approches traditionnelles comme `iptables`.

2.9.3.2 eBPF non privilégié

Parmi les champs d'application de BPF et d'eBPF, il peut être souhaitable de permettre à des processus non privilégiés de charger des programmes eBPF afin de faire des traitements particuliers. Cette approche se justifie d'un point de vue technique puisque le langage eBPF ne permet pas *en théorie* de compromettre le système. Ainsi, seccomp-bpf permet aux processus y compris non privilégiés de mettre en place des politiques de sécurité BPF pour restreindre le processus courant. Similairement, les versions anciennes de Landlock permettaient aux processus non privilégiés de mettre en place des politiques

eBPF. Ce choix technique est particulièrement intéressant dans le cadre des conteneurs puisqu'il leur permet de prendre part dans leur sécurité.

Le langage eBPF est aujourd'hui considéré comme relativement mature. Toutefois, quelques vulnérabilités ont été trouvées dans le vérifieur eBPF, telle que la CVE-2021-3490 [85], qui permet de réaliser des écritures ou des lectures hors limites, pouvant engendrer des exécutions arbitraires de code ou la CVE 2017-9150 [78] pouvant permettre de récuperer des pointeurs kernel, facilitant d'autres attaques subséquentes. Par ailleurs, il est difficile de protéger eBPF contre certains types d'attaques dont les attaques spéculatives, les fuites de pointeur kernel, ... Pour cette raison, une partie de la communauté eBPF, y compris son mainteneur, pense que l'eBPF non privilégié est un cas d'usage qui reste trop risqué pour le noyau, et que ce cas d'usage peut de fait être déprécié. La direction prise par le noyau est donc de restreindre les programmes pouvant charger des programmes eBPF à ceux disposant d'une capabilité (`CAP_BPF`). Cette approche présente un compromis permettant à des utilisateurs non-root de charger du code eBPF tout en ne permettant pas la mise en place de programmes BPF arbitraires par des utilisateurs non privilégiés [26].

Nous pensons que si le chargement de programmes eBPF non privilégiés *arbitraires* peut poser des problèmes de sécurité sévères, permettre aux programmes non privilégiés de charger uniquement des programmes eBPF venant de sources de confiance peut permettre la mise en place de programmes eBPF sans poser de problématiques de sécurité. Cette approche est explorée dans le chapitre 4. Il reste toutefois possible de limiter le chargement de programmes eBPF aux seuls programmes disposant de `CAP_BPF` pour les environnements le souhaitant.

Chapitre 3

SNAPPY

Résumé du chapitre *Dans ce chapitre, nous présentons le design et l'implémentation de notre contribution SNAPPY, un nouveau framework permettant d'installer des politiques de sécurité programmables à grain fin de niveau noyau applicables par namespace, afin d'améliorer la sécurité des conteneurs. Nous détaillons plus particulièrement les mécanismes clefs de SNAPPY : l'isolation de politiques par namespace noyau, l'interface noyau de chargement et l'exécution des politiques au niveau des hooks LSM et enfin, le concept de 'helper dynamique' chargeable à chaud dans le noyau et utilisable par les politiques eBPF. Nous montrons que SNAPPY peut être intégré simplement aux standards existants (OCI, runC, Docker). Nous présentons finalement des cas d'usages illustratifs des capacités de ce framework et nous comparons SNAPPY avec l'état de l'art.*

Contents

3.1 Besoin de politiques de sécurité plus flexibles

3.1.1 Contexte

Nous avons expliqué dans le chapitre 1 que les conteneurs étaient une primitive proposant de nombreux avantages sur les approches concurrentes dont les machines virtuelles, en permettant notamment un déploiement simplifié et une performance extrêmement proche du code natif. Toutefois, nous avons aussi expliqué en chapitre 2 que de par leur design, les conteneurs étaient aussi très vulnérables aux attaques, avec des conséquences qui peuvent être très sérieuses.

Ainsi, améliorer la sécurité des conteneurs pour s'approcher du niveau de sécurité des machines virtuelles est un objectif très attrayant. Cela permettrait aux utilisateurs de conteneurs de bénéficier des avantages de la conteneurisation sans devoir prendre plus de risques en termes de sécurité. Toutefois, puisque les conteneurs utilisent souvent des images vulnérables et que plusieurs conteneurs peuvent être partagés sur une machine alors que la surface d'attaque entre un conteneur et l'hôte est grande (un noyau Linux complet), cette amélioration du niveau de sécurité des conteneurs est très difficile.

De nombreuses lignes de recherche ont tenté d'améliorer la sécurité des conteneurs en développant de nouvelles abstractions logicielles. Ainsi, beaucoup d'efforts ont été fournis pour permettre aux Modules de Sécurité Linux d'être empilables et namespaceables afin de les adapter de manière transparente aux conteneurs. Toutefois, les approches visant à rendre namespaceables les modules LSM sont spécifiques à chaque module et peuvent être très complexes. C'est notamment le cas de SELinux. De telles approches restent par ailleurs insuffisantes puisque par design les modules LSM nécessitent les droits administrateurs pour être utilisés et ne sont donc pas utilisables par des conteneurs non privilégiés pour protéger leurs frontières.

3.1.2 Amélioration de la sécurité des conteneurs

Dans l'optique d'amener la sécurité des conteneurs vers celle des machines virtuelles, nous pensons qu'exploiter plus en profondeur les capacités de programmabilité du noyau afin de permettre de personnaliser très précisément la sécurité des conteneurs est une approche prometteuse. En effet, contrairement à la virtualisation lourde, la conteneurisation n'est basée que sur des primitives Linux existantes. Un conteneur n'est donc qu'un ensemble de processus configurés de manière particulière pour qu'il ne soit pas en mesure de

voir le reste du système. Il est donc possible de profiter très simplement de la programmabilté du noyau pour les conteneurs sans devoir résoudre des problèmes supplémentaires. Les conteneurs semblent donc très adaptés à cette approche.

Une approche pertinente visant à permettre aux processus, dont les conteneurs, d'exploiter les capacités de programmabilité du noyau est Landlock LSM. Dans sa version initiale, Landlock permettait d'appliquer des politiques de sécurité sous forme de programmes eBPF. Il était possible de charger de tels programmes depuis les conteneurs. De telles politiques étaient appliquées aux événements LSM. Nous pensons qu'une telle approche est très prometteuse. Toutefois, Landlock hérite des limitiations d'eBPF limitant la flexibilté d'une telle approche. Par ailleurs, les programmes Landlock ne sont applicables qu'à certains hooks LSMs notamment ceux liés à la gestion des fichiers et du ptracing.

3.1.3 Approche SNAPPY

Nous pensons que permettre y compris aux conteneurs non privilégiés de mettre en place au niveau noyau des politiques de sécurité programmables à grain fin permettrait de les protéger contre un grand nombre d'attaques et serait donc une avancée majeure. De telles politiques doivent ne pas être désactivables pour garantir qu'une fois appliquées, elles protègent effectivement le système. Une telle approche doit être utilisable avec un surcoût en performances minimal pour ne pas affecter les bonnes propriétés de performances des conteneurs.

Ainsi, nous proposons le framework SNAPPY (Safe Namepaceable and Progammable PolicY) qui permet même aux processus non privilégiés dont les conteneurs de mettre en place des politiques de sécurité programmables à grain fin dans le noyau. SNAPPY permet notamment aux processus de limiter précisément leur surface d'attaque, d'appliquer des politiques de sécurité métier et de mitiger des vulnérabilités.

Afin d'atteindre cet objectif, nous proposons une nouvelle namespace orientée sécurité que nous nommons `policy_ns`. Cette namespace est responsable du stockage et de la gestion des politiques de sécurité niveau kernel et par conteneur, de SNAPPY. Il est donc possible d'appliquer des politiques de sécurité à toute namespace SNAPPY. Nos politiques programmables peuvent être écrites en langage eBPF. Afin de contourner les limitations d'eBPF tout en gardant les propriétés de sécurité d'eBPF, nous créons le concept de 'helper dynamique', chargeables à chaud. Afin de mettre en place et d'exécuter de telles politiques, nous créons un nouveau module LSM : SNAPPY.

SNAPPY a été implémenté avec succès. Nous avons testé cette approche à la fois

avec des processus Linux classiques et des conteneurs. Nous avons par ailleurs réalisé des intégrations avec des frameworks existants dont OCI et Dockerfile. Notre prototype est relativement léger, possède de très bonnes propriétés de performance et est en couplage faible avec le reste du système.

3.1.4 Contributions

Les contributions présentées dans ce chapitre sont les suivantes :

— Création d'une namespace noyau orientée sécurité (`policy_ns`) permettant de définir des politiques par namespace attachables à des hooks LSM.
— Une interface noyau permettant à tous les processus, y compris non privilégiés de mettre en place des politiques de sécurité eBPF dans le noyau, attachées à une `policy_ns`.
— Les 'helpers dynamiques' qui peuvent être chargés par l'administrateur à l'exécution et qui permettent de déporter les accès aux structures critiques du noyau et de réaliser les traitements complexes pour le compte des politiques de sécurité eBPF.
— L'intégration de SNAPPY aux moteurs de conteneurisation courants (runc et docker) et aux standards de facto (OCI et Dockerfile).

Contributions de l'auteur sur cette thématique

— SNAPPY : programmable kernel-level policies for containers. **Maxime Bélair**, Sylvie Laniepce, and Jean-Marc Menaud. In Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC '21). Mars 2021, Gwangju / Virtual, South Korea. DOI : `https://doi.org/10.1145/3412841.3442037` [12]
— Procédé de sécurisation d'un appel système, procédé de mise en place d'une politique de sécurité associée et dispositifs mettant en œuvre ces procédés. **Maxime Bélair**, Sylvie Laniepce. N° de dépôt : 20 2095. Déposé le 20/05/2020 (numéro référence extension PCT FR2021050860).
— Code source de SNAPPY : `https://github.com/Orange-OpenSource/SNAPPY`.

Organisation du chapitre Ce chapitre est organisé comme suit : nous présentons le design global de SNAPPY en section 3.2.1. Nous présentons la nouvelle namespace introduite par SNAPPY (`policy_ns`) en 3.2.2. Nous présentons ensuite le design des politiques de sécurité utilisées par SNAPPY en section 3.2.3, le design des helpers "dynamiques" utilisés par SNAPPY en 3.2.4 et nous réalisons l'analyse de sécurité de SNAPPY en 3.2.5.

Nous discutons de l'intégration de SNAPPY avec des primitives courantes de Linux en section 3.3. Nous présentons des cas d'usage de SNAPPY en section 3.4. Nous terminons ce chapitre en comparant l'approche SNAPPY à l'état de l'art en 3.5 et en présentant ses limites en section 3.6.

3.2 Design et implémentation

3.2.1 Design Global

Dans cette sous-section, nous présentons le design de SNAPPY, un framework de sécurité optimisé pour les conteneurs permettant d'appliquer des politiques de sécurité programmables de niveau noyau. De manière synthétique, SNAPPY crée un nouvel espace de nommage Linux orienté sécurité "`policy_ns`" et permet d'appliquer des politiques eBPF aux hooks LSM pour protéger individuellement des namespaces `policy_ns`. SNAPPY pallie les limitations d'eBPF par l'introduction du concept de "helper dynamique" qui est une amélioration du concept existant de "helper statique". Puisque l'approche SNAPPY est basée sur les espaces de nommage Linux, cette approche peut être utilisée pour protéger tout type de machine basée sur Linux. Si le cas d'usage des conteneurs est largement détaillé dans cette thèse, il convient de comprendre que ce n'est pas le seul cas d'application possible et SNAPPY peut également être utilisé pour protéger n'importe quel(le) type de (hiérarchie de) processus comme il sera précisé dans le chapitre 6.2.

SNAPPY permet à tous les processus d'appliquer des politiques de sécurité SNAPPY pour protéger des opérations LSM de sa propre namespace. Il est par exemple possible de vérifier une propriété de sécurité quand un fichier est ouvert, quand un paquet réseau est envoyé, ... Les politiques eBPF sont mises en place en les chargeant dans le noyau à travers une interface noyau créée à cet effet. Afin de rendre les interfaces de SNAPPY utilisables par les conteneurs, ces interfaces noyau sont automatiquement montées à l'intérieur des conteneurs. Les politiques de sécurité sont alors chargées dans la `policy_ns` du processus dans un champ contenant l'ensemble des politiques applicables pour ce hook LSM. Une fois mise en place, la politique est appliquée à ladite namespace et à ses descendantes pour des raisons de sécurité détaillées dans la prochaine sous-section. Ainsi, à chaque fois qu'une opération LSM est effectuée, toutes les politiques de sécurité chargées de vérifier la validité de cette opération sont exécutées. L'opération est réalisée uniquement si *toutes* les politiques ayant un avis sur l'opération la valident. Dans le cas contraire, l'opération

est refusée et des traitements supplémentaires peuvent éventuellement être mis en place (alerte au SIEM, logging noyau, mise en sécurité, ...).

La figure 3.1 montre les étapes principales exécutées lors d'un appel système : lors de l'exécution d'un appel système, une (ou plusieurs) opérations LSM peuvent être déclenchées. Un code est alors exécuté par le gestionnaire LSM. Celui-ci appelle tous les modules LSM qui monitorent le hook protégeant l'opération déclenchée, dont le gestionnaire du LSM SNAPPY. Ce dernier exécute toutes les politiques eBPF de la hiérarchie de namespaces du processus courant, c'est à dire celui qui a déclenché l'opération. Ces programmes eBPF ont accès à des helpers dynamiques responsables de réaliser les traitements complexes et de l'accès sûr aux structures noyau. Si une politique SNAPPY refuse une opération, cette dernière est globalement refusée et l'appel système va échouer.

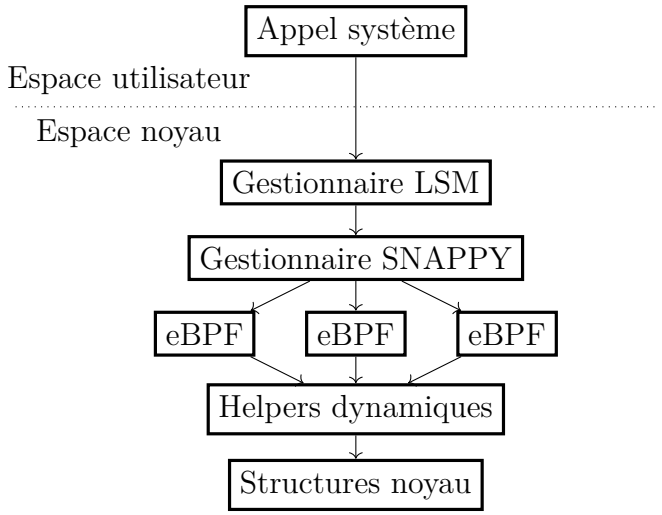


FIGURE 3.1 – Schéma global de SNAPPY

3.2.2 Espace de nommage Linux `policy_ns`

3.2.2.1 Justification du choix technique de la namespace

Il existe trois approches principales pour appliquer des politiques de sécurité à des sous-ensembles du serveur :

— utiliser un "véritable" espace de nommage Linux, c'est à dire une abstraction de namespace fournie par le noyau sur le modèle des namespaces existantes ;

— Utiliser une émulation d'espace de nommage Linux, c'est à dire recréer fonctionnellement une abstraction équivalente à une namespace noyau mais sans utiliser

les primitives fournies par le noyau ;

— Utiliser un filtrage par hiérarchie de processus, c'est à dire parcourir l'exécution de ses ancêtres dans l'arbre des processus et appliquer au processus courant toutes les politiques des processus ascendants).

Nous justifions ici que l'utilisation d'une "véritable" namespace est le meilleur choix technique pour notre cas d'usage. Nous détaillons par la suite le processus de développement de cette namespace orientée sécurité.

La plupart des frameworks de sécurité présentés dans cette thèse n'utilisent pas de véritable namespace de sécurité. Les protections offertes sont donc soit par hiérarchie de processus (par exemple Landlock, Security Namespace), soit utilisent une *émulation* de namespace (par exemple AppArmor, SELinux). Si ces deux approches peuvent fonctionner, elles possèdent des limitations résumées par la table 3.1.

	Véritable NS	NS émulée	Hiérarchie de proc.
Simplicité de développ.	●	○	●
Performance	●	○	◐/ ●
Gestion des démons sys.	●	◐	○
Facilité integrat. Linux	○	◐	●

TABLE 3.1 – Comparaison entre les différentes approches d'isolation de niveau noyau

La table 3.1 présente les principales différences entre ces trois approches. On constate que puisque la primitive namespace est déjà implémentée et très mature, la création d'une nouvelle namespace de sécurité est un mécanisme très simple à développer, maintenir et auditer. Similairement, un mécanisme basé sur la hiérarchie de processus peut être implémentée de manière très simple, en appliquant les politiques à un processus de base et en vérifiant pour toute la hiérarchie de processus si des politiques de sécurité s'appliquent à ce processus. Au contraire, les émulations de namespace créent souvent des codes assez complexes, difficiles à comprendre et à auditer. De plus, elles dupliquent dans le noyau la fonctionnalité namespace, déjà implémentée, augmentant la surface d'attaque. En termes de performances, c'est à dire d'overhead à l'exécution de politiques de sécurité, l'utilisation d'une namespace donne de bonnes propriétés de performances puisqu'il suffit d'exécuter les politiques de la namespace courante et celles de ses ancêtres éventuels. L'utilisation d'une hiérarchie de processus suit le même raisonnement en faisant exécuter les politiques applicables au processus courant et à ses ancêtres. Toutefois, l'utilisation d'une hiérar-

chie de processus peut donner un overhead légèrement supérieur puisque la profondeur de l'arbre des processus est souvent largement plus grande que celle de l'arbre des namespaces. Ainsi, on peut facilement avoir une profondeur d'arbre de processus supérieure à 10 alors que l'arbre des namespaces sera généralement en dessous de 3 ou 4. L'overhead d'une namespace émulée est le pire cas puisqu'elle nécessite de chercher et de faire des mappings entre des structures externes pour trouver la liste de politiques à exécuter.

Ainsi le choix d'utiliser une namespace de sécurité semble un choix technique pertinent. Il possède toutefois quelques limitations, à prendre en compte. En effet, puisque la création d'une nouvelle namespace (par exemple via `unshare` ou `clone`) est basée sur un flag, le nombre de namespace qui peut être créé dans un noyau Linux est limité. Dans le cadre de SNAPPY, nous utilisons le dernier flag aujourd'hui disponible. Il convient donc de prouver à la communauté Linux que l'approche namespace est bien la bonne approche pour envisager une acceptation dans le noyau. Il faut toutefois noter que l'appel système `clone3()` qui permet également de créer des nouvelles namespaces, est déjà disponible dans le noyau et n'est pas affectée par ce problème [41]. Ce problème reste donc acceptable en pratique.

3.2.2.2 Design et implémentation

L'approche retenue pour gérer les politiques de sécurité a donc été de créer une véritable namespace de sécurité `policy_ns`. Comme pour toute namespace, il est possible de créer un processus dans une nouvelle namespace à l'aide des appels systèmes `fork()` ou `clone()` et de changer un processus de namespace à l'aide de `unshare()` ou `setns()`. Le flag utilisé pour `policy_ns` est `0x00400000`, jusqu'à présent ignoré dans le noyau Linux [45]. Afin de permettre l'utilisation de hiérarchies de namespaces, `policy_ns` est créée comme un arbre. Ainsi, quand une nouvelle namespace est créée, celle-ci devient fille de la namespace du processus l'ayant créée. Si un nouveau processus est créé sans création de nouvelle namespace, le nouveau processus est simplement placé dans la namespace du processus parent. Cette hiérarchie de namespace est réalisée techniquement en rajoutant à l'abstraction `policy_ns` un champ "parent". Quand une nouvelle namespace est créée, ce champ est initialisé avec la namespace l'ayant créée. Ainsi les namespaces forment un arbre dont une branche est créée à chaque fois qu'une namespace est créée. Il est donc possible de récupérer très simplement les ancêtres d'une namespace en parcourant récursivement le champ `parent` de cette namespace. Quand un hook LSM est déclenché, les politiques de *toutes* les namespaces ancêtres liées à ce hook sont appliquées. L'accès est

donné si et seulement si toutes les politiques valident cet accès.

Dans l'optique de stocker les politiques de sécurité applicables à une opération donnée pour une namespace donnée, chaque namespace `policy_ns` contient sa propre liste de politiques de sécurité devant être appliquées pour les processus dans la namespace donnée. Les politiques sont mises en place pour la namespace et ses éventuelles namespaces descendantes, c'est à dire pas pour tout le système. C'est particulièrement utile pour les conteneurs qui peuvent ainsi s'appliquer des politiques de sécurité différentes de celles des autres conteneurs. Ainsi, pour appliquer des politiques spécifiques pour un (ensemble de) processus, il suffit de le(s) créer dans une nouvelle namespace `policy_ns`. De la même manière, quand un nouveau processus est créé, il peut être soit créé dans la namespace courante soit dans une nouvelle namespace fille de la namespace actuelle. Dans le cadre de SNAPPY, l'administrateur est en mesure de choisir l'ensemble des hooks pouvant activer des politiques de sécurité. Cela permet d'utiliser SNAPPY pour n'importe quel hook sans souffrir de l'impact sur les performances qui serait généré si SNAPPY activait automatiquement la protection pour tous les hooks.

Le fonctionnement de la namespace `policy_ns` est illustré par la figure 3.2. Elle montre que chaque processus possède à travers la structure `nsproxy` un pointeur vers la namespace `policy_ns` qui lui est applicable, que chaque namespace possède un champ "state" permettant de mettre en place des politiques à états (stateful) et un champ pointant vers la namespace parent. Finalement chaque namespace contient une liste de politiques de sécurité renseignées par hook LSM.

Pour garantir que SNAPPY ne puisse pas être utilisé par un utilisateur malveillant pour compromettre le système, nous rendons impossible de supprimer ou de modifier une namespace pendant la totalité du cycle de vie des conteneurs qu'elle protège. Cela signifie que si un utilisateur veut supprimer ou modifier des politiques de sécurité pour un conteneur, il devra terminer l'exécution dudit conteneur et le redémarrer avec les nouvelles politiques de sécurité. Ceci est bénéfique sur le plan de la sécurité car cela empêche qu'un conteneur compromis puisse supprimer ses politiques de sécurité pour effectuer une élévation de privilèges. Par ailleurs, cette mesure ne coûte pas beaucoup en termes de performances puisque les conteneurs sont généralement très rapides à relancer. Au contraire, il est possible d'ajouter de nouvelles politiques de sécurité à tout moment du cycle de vie de la namespace. De telles politiques ne peuvent pas ajouter de nouveaux droits sur les conteneurs puisque les politiques sont empilées et il suffit qu'une politique refuse l'accès pour que celui-ci soit globalement refusé. Ainsi, ajouter une politique ne

peut que restreindre ou laisser inchangé l'ensemble des droits des processus appartenant à la namespace. Par ailleurs, puisque les politiques d'une namespace s'appliquent aussi aux éventuelles namespaces filles, ces dernières héritent des politiques de tous ses ancêtres empêchant toute élévation de privilège à travers l'utilisation de nouvelles namespaces.

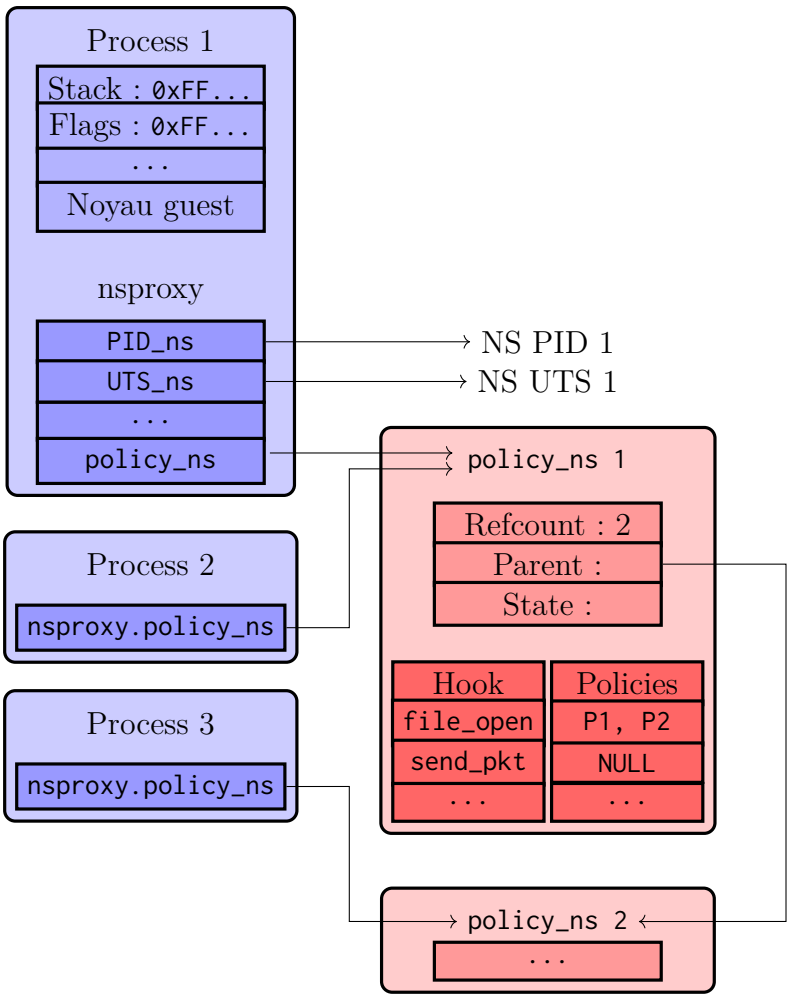


FIGURE 3.2 – Exemple de `policy_ns`

3.2.2.3 Adaptation aux moteurs de conteneurisation système

Bien qu'en général le champ `parent` d'une `policy_ns` corresponde à la namespace du parent du processus qui l'a créée, il existe une exception à cette règle. En effet, un certain nombre de moteurs de conteneurisation dont Docker créent des conteneurs à travers des démons système. Dans de tels scénarios, le conteneur nouvellement créé est en réalité un

descendant du démon système l'ayant créé. Par exemple, dans le cas de Docker le conteneur est en réalité créé à travers containerd-shim, un descendant de ce démon système [8, 53]. Ainsi si nous ne faisions rien, le nouveau conteneur hériterait des politiques liées à la `policy_ns` de containerd-shim et non de celles de la `policy_ns` du processus l'ayant créé. Ceci n'est pas acceptable puisque containerd-shim n'est pas un descendant du processus ayant créé le nouveau conteneur. C'est pourquoi, nous forçons le nouveau conteneur à être créé dans une nouvelle namespace fille de la namespace du processus qui l'a *logiquement* créé. Cette modification de comportement a été implémentée grâce à des modifications très mineures dans le moteur de conteneurisation : en récupère la namespace du processus courant qui demande la création du nouveau conteneur, on modifie la spécification du runtime OCI du conteneur et en ajoutant au fichier JSON de la spécification du runtime OCI du nouveau conteneur créé, un objet `policy_ns` référant à un enfant de la namespace du "parent logique" du conteneur créé. Cette modification à été implémentée en 12 lignes de code et rend SNAPPY compatible avec la spécification de runtime OCI qui est la spécification de référence pour les images de conteneurs. Cette approche est trivialement adaptable aux autres moteurs de conteneurisation.

Par conséquent, notre arbre de namespaces `policy_ns` est totalement indépendant de l'arbre des processus. Cette décorrélation permet à notre design de s'appliquer dans toutes les situations, y compris dans le cas de la virtualisation imbriquée.

3.2.2.4 Politiques à états

De manière bien étudiée dans la littérature [30], les processus ont généralement des comportements différents à l'initialisation et pendant leur boucle principale. En effet dans de nombreux cas, la liste des comportements légitimes est plus grande à l'initialisation car le processus a besoin d'accéder à beaucoup de fonctions pour mettre en place les entités (données, connexions, fichiers, ...) qui sont utilisées dans la boucle principale. Il est donc souhaitable de pouvoir appliquer des politiques différenciées entre cette phase d'initialisation et la boucle principale. Ce comportement est permis par SNAPPY grâce au champ `state` de `policy_ns`. Ce champ est un entier auquel il est possible de faire correspondre les différentes phases du cycle de vie du conteneur, par exemple initialisation = 0, boucle principale = 1, ... Il possible de n'appliquer des politiques eBPF qu'à partir d'un certain état, par exemple ici à partir de 1 pour ne pas appliquer les politiques à l'initialisation. Quand une namespace est initialisée, le champ état est initialisé à l'état zéro. Il est possible de faire passer une namespace à l'état suivant à l'aide d'un helper eBPF

dédié (HELPER_STATE). Pour des raisons de sécurité, la transition inverse est impossible. Ainsi par design, il est possible d'appliquer des politiques différenciées aux différentes phases de vie du conteneur. De manière générale, on s'attend à ce que ces politiques soient de plus en plus restrictives.

En conclusion, notre design permet de créer des politiques par namespace et de s'assurer que sous aucune circonstance un processus ne soit en mesure d'obtenir des droits plus élevés qu'à un temps précédent ou que ceux de l'un de ses ancêtres. Cette propriété est fondamentale d'un point de vue de la sécurité car elle garantit que les politiques censées être appliquées à une namespace sont toujours appliquées, garantissant ainsi la sécurité du système.

3.2.3 Politiques de sécurité SNAPPY

3.2.3.1 Justification de l'utilisation d'eBPF

Les politiques utilisées par SNAPPY sont écrites dans le langage eBPF pour les raisons suivantes :

— Les approches basées sur le code permettent de définir de manière finement grainée une logique utilisateur et proposent une versatilité plus grande que les approches basées sur les règles et à plus forte raison, que les approches basées sur la configuration.
— eBPF permet d'appliquer des politiques directement dans le kernel. Elles sont donc isolées des processus/conteneurs qu'elles contrôlent qui sont eux en espace utilisateur.
— L'application de politiques purement dans le kernel, exécutées lors d'événements LSM permet par ailleurs d'éviter tout changement de contexte (context-switch) [116], évitant ainsi les surcoûts en performances que ceux-ci génèrent.
— Le langage eBPF est un langage mature, bien établi et largement audité par la communauté Linux. Il possède de très bonnes propriétés de sécurité et de performances.

Nous avons implémenté un nouveau type eBPF pour nos politiques. Puisqu'il est fondamental que SNAPPY ne puisse pas être utilisé pour compromettre la sécurité du système c'est à dire gagner de nouveaux privilèges, faire crasher le système via un kernel panic, ..., ce nouveau type eBPF a été durci au maximum. En plus d'interdire tout accès direct à des structures noyau, nous interdisons de déréférencer le pointeur "contexte" qui

contient les arguments nécessaires au traitement de la politique et qui correspondent généralement aux arguments du hook correspondant. En effet, des accès incorrects à cette structure, y compris en lecture-seule pourraient avoir des conséquences sur la sécurité du système, par exemple en permettant de faire fuiter des pointeurs kernels facilitant de futures attaques [20].

Pour des raisons de sécurité mais aussi de flexibilité et de sécurité, le type eBPF utilisé par SNAPPY n'utilise qu'un seul helper statique, présenté en détail en sous-section 3.2.4. Cela permet de limiter le plus possible la surface d'attaque tout en permettant un comportement plus flexible que les helpers classiques. Par exemple, le type eBPF XDP peut faire appel à 15 helpers statiques, la surface d'attaque de ce type eBPF ne peut donc pas être diminuée en dessous de cette limite.

3.2.3.2 Mise en place de politiques de sécurité SNAPPY

Les politiques eBPF SNAPPY peuvent être mises en place pour protéger les hooks LSMs. Afin d'être le plus générique et versatile possible, nous permettons de charger des politiques pour protéger tous les hooks LSMs. Il est à noter que les hooks LSM sont considérés comme une API kernel instable, c'est-à-dire qu'ils peuvent être modifiés voire supprimés sans assurer de rétrocompatibilité entre deux versions du kernel. Nous utilisons donc une couche d'abstraction entre les hooks LSM et les programmes SNAPPY, les "hooks SNAPPY". Dans l'implémentation actuelle cette couche d'abstraction appelle directement les hooks LSM correspondant. On note identiquement les hooks LSM et SNAPPY, mais respectivement en minuscule et en majuscule (par exemple LSM : `file_open`, SNAPPY : `FILE_OPEN`).

Les politiques eBPF peuvent être écrites comme n'importe quel programme eBPF c'est à dire à partir d'un sous-ensemble du C compatible avec les limitations de eBPF (pas de boucles infinies, pas d'arithmétique pointeur, pas de sauts remontant dans le code, ...). La compilation de ce programme eBPF peut notamment être faite à l'aide du compilateur llvm. Il est évidemment possible d'écrire les programmes directement en bytecode eBPF mais cette approche est plus complexe et donc généralement moins intéressante.

Après compilation, nous permettons à tous les utilisateurs, y compris les utilisateurs non-administrateurs de charger ces politiques dans le noyau, comme illustré par la figure 3.3. Le programme eBPF compilé peut être chargé à travers une interface noyau que nous avons proposée pour charger le programme au niveau du hook LSM correspondant. Nous proposons également des méthodes de chargement de plus haut niveau présentées

en sous-section 3.3.2. Par exemple, les politiques chargées dans l'interface noyau `/sys/kernel/security/snappy/policies/FILE_OPEN` sont appliquées au hook SNAPPY `FILE_OPEN` correspondant au hook LSM `file_open`. En interne, ce chargement utilise l'appel système `bpf()`. Au chargement de la politique, le vérifieur eBPF vérifie que le programme eBPF est correct, c'est à dire qu'il respecte bien les contraintes de sécurité d'eBPF qui garantissent que le programme ne peut pas attaquer le système. Si le programme est déclaré valide par le vérifieur, il est chargé dans la `policy_ns` du processus qui a chargé la politique, dans la structure correspondant au hook LSM SNAPPY à protéger. Si des politiques étaient déjà présentes pour ce hook dans cette namespace, ces programmes sont simplement empilés. Le programme est immédiatement fonctionnel : à chaque fois qu'un processus de cette namespace ou une namepsace descendante réalise un appel système qui déclenche ce hook LSM, cette politique de sécurité est exécutée, ainsi que les éventuelles autres politiques appliquées à ce hook pour cette hiérarchie de namespace et l'exécution de l'appel système est conditionnée par le résultat de cette (ces) politique(s).

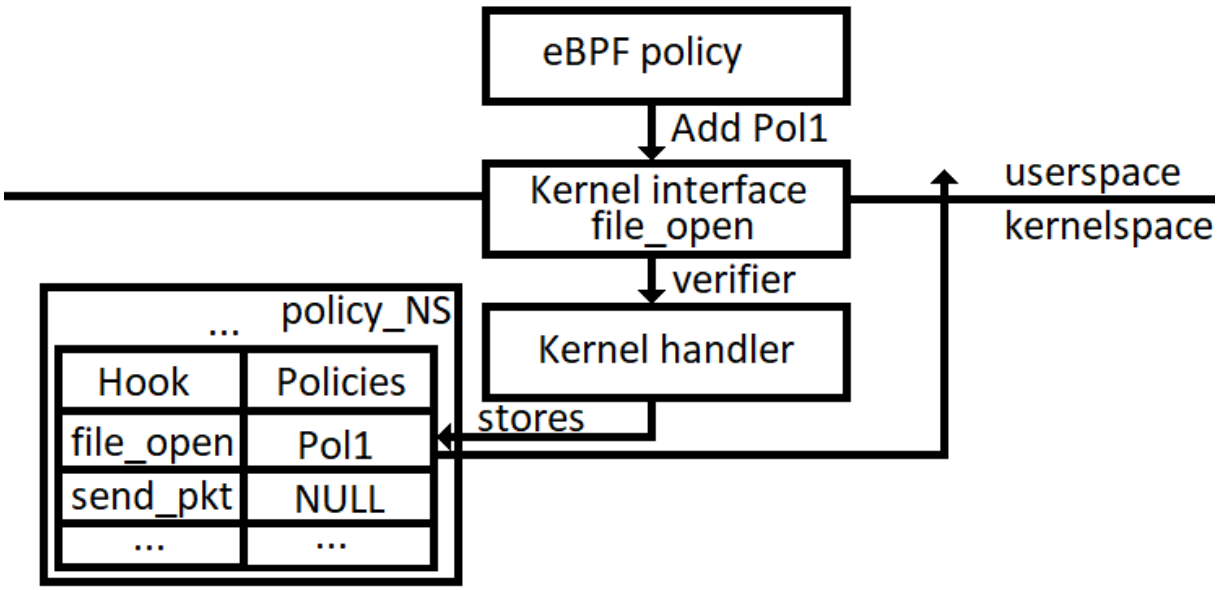


FIGURE 3.3 – Chargement de politiques eBPF dans la namespace

Comme indiqué en section 2.9.3, une partie de la communauté Linux s'oppose à l'utilisation de nouveaux types eBPF non privilégiés dans le noyau et un débat est en cours sur la pertinence du cas d'utilisation 'eBPF non privilégié' pour le futur [26]. Toutefois, puisque nous définissons un nouveau type eBPF volontairement très limité, nous considé-

rons que la sécurité de notre type eBPF est élevée et potentiellement meilleure que certains types existants comme XDP. Nous pensons qu'un tel type eBPF peut raisonnablement être intégré dans le système sans en compromettre la sécurité. Toutefois, si une opposition à une telle intégration devait apparaitre, nous proposons de rendre configurable l'ensemble des processus pouvant charger des politiques eBPF pour le type SNAPPY à l'aide d'un paramètre de configuration noyau. Il est possible d'autoriser tous les processus à charger une politique eBPF, seuls les processus disposant de la capabilité `CAP_BPF` ou seulement les processus disposant des droits administrateur `CAP_SYS_ADMIN`. Enfin, en section 4, nous montrons qu'il est possible d'intégrer à SNAPPY des mécanismes d'authentifications qui permettent de garantir que les politiques SNAPPY viennent d'une source de confiance. Un tel mécanisme d'authentification permet de limiter les politiques installables à des politiques validées par une source de confiance, ce qui exclut le cas du code arbitraire.

3.2.4 Helpers dynamiques

3.2.4.1 eBPF et helpers statiques

L'infrastructure présentée dans les dernières sous-sections permet d'exécuter des politiques de sécurité programmables et applicables de manière indépendante à des espaces de nommage. Toutefois, telles que définies jusqu'ici, les politiques restent assez peu flexibles puisqu'elles héritent des fortes contraintes du langage eBPF, a fortiori puisque nous limitons fortement les opérations réalisables par les programmes utilisant notre type eBPF. Il est donc nécessaire de mettre en place des mécanismes pour effectivement avoir des politiques à grain fin et non pas simplement des switchs accepter/refuser une opération.

Comme indiqué en section 2.9.3.1, la manière classique de permettre à un code eBPF de réaliser des opérations supplémentaires est d'utiliser des "helpers eBPF". Ces helpers consistent en du code implémenté dans le kernel et sont responsables de faire des opérations complexes ou nécessitant d'accéder à des structures critiques pour le programme eBPF les appelant. Par exemple `bpf_map_lookup_elem` permet de regarder une valeur dans une map eBPF. Dans la suite de ce document, nous appelons ces helpers "helpers statiques" pour les différencier des "helpers dynamiques" que nous introduisons dans cette thèse. Nous qualifions ces helpers de statiques car il n'est pas possible de modifier ou de rajouter de helpers statiques pour un noyau donné [1].

1. à moins de manuellement modifier le code du noyau, de le recompiler et de redémarrer le serveur avec le nouveau noyau. Une telle approche est complexe, peut générer des erreurs qui peuvent résulter en des paniques noyau (kernel panic) ou affaiblir la sécurité du serveur. Une telle approche n'est donc pas

Les programmes eBPF peuvent appeler des helpers statiques de manière similaire à des appels de fonctions. L'exécution de telles politiques n'engendre pas de changement de contexte puisque tout est réalisé en espace noyau, permettant d'exécuter ces helpers statiques sans surcoût et donc de conserver les propriétés de performances d'eBPF. Puisque ces helpers consistent en du code kernel, seul un petit nombre d'helpers est disponible, limitant l'utilisation de ces helpers à un petit nombre de traitements précis.

Ainsi, l'approche des helpers statiques n'est pas satisfaisante pour notre cas d'usage. En effet, puisque les politiques SNAPPY sont potentiellement très diverses, une telle approche nécessiterait d'implémenter et d'auditer un *très* grand nombre d'helpers. Cela résulterait en une augmentation importante de la surface d'attaque du noyau, ce qui n'est pas souhaitable. Par ailleurs, une telle approche aurait beaucoup de mal à s'adapter à l'évolution des cas d'usage de SNAPPY et nécessiterait donc de constamment devoir implémenter de nouveaux helpers et recompiler le noyau avec ces nouveaux helpers.

3.2.4.2 Design des helpers dynamiques

Ainsi, nous créons dans cette thèse le concept de "helper dynamique". Ces helpers contrastent avec les helpers statiques car ils peuvent être ajoutés à chaud dans le noyau, c'est-à-dire sans avoir besoin de modifier, recompiler le noyau et de redémarrer la machine pour satisfaire à des besoins de sécurité précis. Notre approche permet donc à l'administrateur de générer *facilement* des helpers dynamiques et de les charger dans le kernel à chaud. Dès que les helpers dynamiques sont chargés dans le noyau, ils deviennent utilisables par les politiques de sécurité eBPF. Les helpers dynamiques peuvent contenir une ou plusieurs fonctions support pour le code eBPF. Les fonctions support du helper dynamique possèdent le prototype `int dynfun(struct snappy_ctx ctx*, void** args)` où `ctx` permet d'accéder aux informations du hook LSM exécuté (par exemple, accéder à un fichier avec `file_open` donne accès à un pointeur sur le fichier accédé) et `args` correspond aux éventuels arguments transmis par le programme eBPF.

réaliste en production.

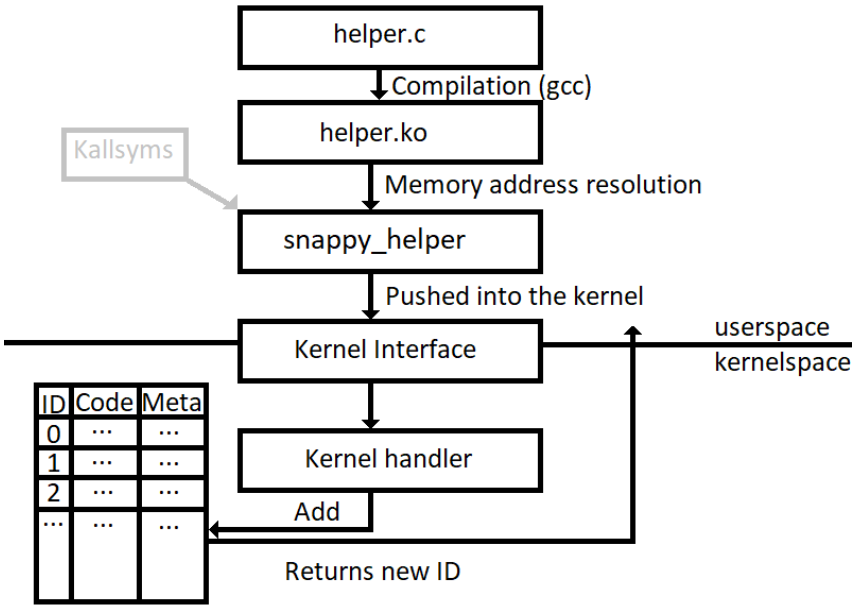


FIGURE 3.4 – Compilation et chargement de helpers dynamiques

Les helpers dynamiques peuvent être construits aussi simplement que des modules noyau, c'est à dire comme un programme C ayant accès à toutes les fonctions noyau exposées par le kernel.

Le design de SNAPPY permet de charger à la discrétion de l'administrateur tout type de helpers dynamiques, pour gérer des traitements précis ou des cas plus généraux. L'approche qui semble la plus prometteuse est la création de helpers génériques qui traitent toutes les opérations usuelles pour une famille de cas d'usages par exemple la gestion des fichiers, la gestion du réseau, ... Grâce à de tels helpers dynamiques, il est possible avec un petit nombre d'helpers de gérer un très grand nombre de cas d'usages et donc d'obtenir des politiques à grain très fin sans avoir besoin de mettre en place un grand nombre d'helpers. Le design et l'implémentation des tels helpers génériques sont discutés plus en détail en section 4.2.6 pour le cas d'usage SecuHub.

Nous avons fait le choix technique de charger ces helpers dynamiques dans la zone mémoire `vmalloc` et non la mémoire noyau `kmalloc` classique. Ce choix permet de pouvoir charger des helpers indépendamment des limites mémoire du noyau (512 MO) [117]. Puisque le noyau prend également une partie de cette zone mémoire, cela aurait très largement limité le nombre de helpers si nous avions chargé ces helpers dans la mémoire du noyau. En allouant nos helpers dynamiques à l'aide de `__vmalloc`, nous obtenons des don-

nées éloignées de plusieurs téraoctets des adresses noyau, voire pétaoctets sur des tables de pages à 5 niveaux. Cependant, puisque les noyaux `x86`, y compris `x86-64` utilisent le code model `small` [69], il n'est possible au niveau assembleur avec des appels classiques (appels directs, générés par défaut par le gcc) que de faire référence à des symboles à au plus 2 GO de l'adresse courante. Ainsi, notre code n'est pas en mesure de faire référence aux symboles noyau à l'aide d'appels directs. Pour éviter ce problème, nous forçons le code généré à utiliser exclusivement des appels de fonctions indirects à l'aide des options de compilation de gcc `-mforce-indirect-call -mindirect-branch=keep -fpie`. Nous générons dans le binaire une table globale d'offsets (section `.got` pour Global Offset Table) contenant l'adresse 64 bits de tous les symboles auxquels le helper dynamique à besoin d'accéder.

Ainsi, lors d'appels à des symboles externes, le binaire génère un code où les fonctions sont appelées de manière indirecte comme montré dans le listing 3.1. Les couleurs utilisées dans la figure regroupent logiquement les informations. Dans ce listing, l'instruction à l'adresse `0x400170` est un appel indirect à un offset de `0x1e82` du pointeur sur la prochaine instruction (`%rip`) [70], ce qui correspond à l'adresse `401ff8`. Ainsi, le programme saute à l'adresse située à 8 octets du symbole GoT, c'est à dire le deuxième pointeur de la table globale d'offsets.

```
400170 :   ff 15 82 1e 00 00    callq *0x1e82(%rip)   # 401ff8 <.got+0x8>
```

Code 3.1 – Exemple d'appel indirect assembleur

3.2.4.3 Génération et chargement

La génération et le chargement de helpers dans le noyau sont détaillés dans la figure 3.4. Toute la procédure de compilation est réalisée par un binaire de compilation `build.sh` que nous fournissons par exemple `./build.sh -t helperName`. Les helpers sont tout d'abord compilés en tant que module noyau. Afin de rendre ce code exécutable, nous résolvons alors les symboles noyau avant leur chargement dans le noyau. Techniquement, cette édition de liens est réalisée en récupérant les adresses noyau à l'aide de `kallsyms`. La randomisation des espaces d'adressage noyau KASLR (Kernel Adress Space Layout Randomisation) [24] est gérée simplement en enlevant à chaque entrée de la table d'offsets, l'offset noyau lié à ce démarrage (récupéré à partir d'une adresse connue (`_text`) qui changera au prochain démarrage et donc qui ne doit pas être conservé. Au chargement dans le noyau, l'offset est rajouté aux symboles. Cette résolution est rapide car il suffit de changer la valeur

une fois par symbole, même s'ils sont appelés un grand nombre de fois grâce à la table d'offsets. Une fois les adresses résolues, il est possible de charger le helper dynamique [2]. Le chargement des helpers dynamiques s'effectue en chargeant le code eBPF à travers une interface kernel que nous avons prévue à cet effet (`/sys/kernel/security/snappy/snappy_load`). Les données chargées dans cette interface comprennent le code du helper ainsi que des métadonnées (nom du helper, hash du helper, taille et offset de la table globale d'offsets, nombre et offsets des points d'entrées). Les helpers sont récupérés par SNAPPY et stockés dans le noyau via une structure associée. Dès que les helpers sont chargés, ils peuvent être appelés par les politiques eBPF.

3.2.4.4 Utilisation des helpers genériques

Comme évoqué dans la section 3.2.3, notre type eBPF ne permet que l'utilisation d'un seul helper statique eBPF. Ce helper nommé`snappy_dynamic(int lib_id, int fn_id , void** args)` fait office de multiplexeur pour appeler les différents helpers dynamiques. Le fonctionnement de ce helper est illustré par la figure 3.5. Ainsi, il est possible d'appeler un helper dynamique avec `snappy_dynamic` à l'aide de son doublet `(helper_id, fn_id)` et un paramètre `args` permet de passer des arguments supplémentaires au helper dynamique pour faire des traitements supplémentaires. En plus de ces arguments, le helper eBPF dynamique a accès au contexte d'appel, c'est à dire les arguments tels que transmis par le hook eBPF.

2. Il aurait été possible de réaliser la résolution et la gestion des adresses mémoires directement dans le noyau en faisant des modifications sur le chargeur de module noyau pour permettre de charger des helpers dynamiques. Ce choix aurait permis une intégration plus unifiée avec l'environnement Linux et un déport d'une partie des procédures dans le kernel, simplifiant la procédure en espace utilisateur. Toutefois, dans l'optique de faire un code couplé le plus faiblement possible avec le noyau, nous n'avons pas suivi cette piste et avons utilisé le chargeur personnalisé, comme présenté ici.

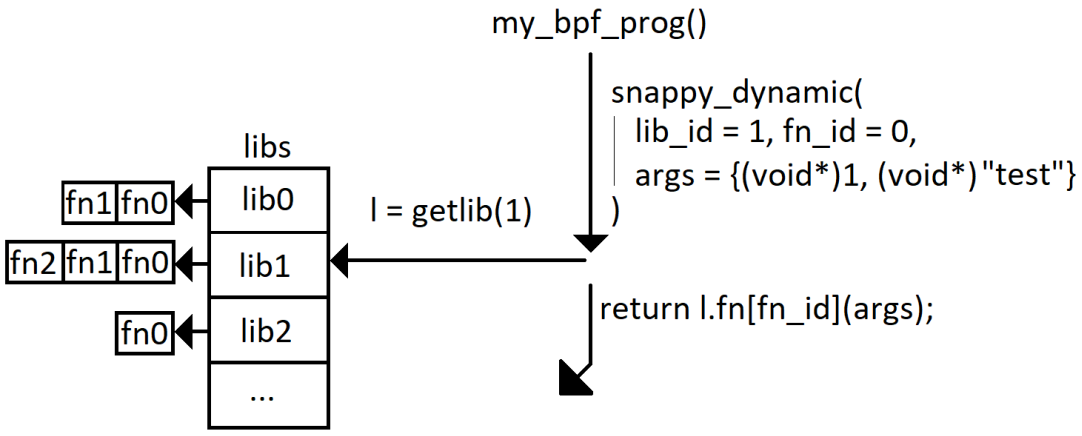


FIGURE 3.5 – Appeler un helper dynamique depuis du code eBPF

3.2.5 Analyse de sécurité

Dans cette section, nous analysons la sécurité du design de SNAPPY. Dans cette optique, nous faisons l'hypothèse suivante :

Hypothèse 1. *L'implémentation de SNAPPY et des mécanismes internes utilisés par SNAPPY (eBPF, LSM, namespaces) ne contiennent pas de vulnérabilité exploitable.*

L'hypothèse 1 semble raisonnable puisque :

— L'implémentation de SNAPPY est relativement simple (environ 750 lignes de code)

— SNAPPY est en couplage faible avec le système, son code peut donc facilement être audité

— SNAPPY utilise principalement des briques logicielles très connues et largement auditées. La probabilité que ces frameworks contiennent une vulnérabilité exploitable à un instant donné est donc relativement faible.

Nous montrons que sous cette hypothèse, le design de SNAPPY est sûr en montrant que SNAPPY ne permet pas à un processus de compromettre plus que lui-même :

— SNAPPY permet à tous les processus, y compris non-privilégies de charger des politiques eBPF vers le noyau. Cependant, ces politiques doivent nécessairement être associées à leur *propre* namespace `policy_ns`. Un processus n'est pas en mesure de créer des politiques qui seraient appliquées en dehors de sa namespace Par ailleurs, de par le design d'eBPF, il n'est pas possible d'utiliser ces politiques pour attaquer le système. Le processus n'est donc pas en mesure d'interférer avec le reste du système. Ainsi, quand un processus qui n'est pas de confiance est exécuté, il suffit de le

confiner dans une nouvelle namespace `policy_ns` pour l'empêcher d'interférer avec la namespace du processus courant puisque les politiques d'une namespace parent sont aussi appliquées aux namespaces filles mais que la réciproque est fausse. Ainsi, un processus est seulement en mesure de s'appliquer à lui-même de "mauvaises" politiques de sécurité pouvant bloquer des comportements légitimes des processus de sa namespace, ce qui peut dans le pire des cas résulter en un Déni de Service (DoS). Ceci ne constitue toutefois pas un problème de sécurité puisque le processus est déjà en mesure d'effectuer un déni de service sur sa propre namespace par d'autres moyens plus simples tel que `pkill` ou `rm -rf`.

— Puisque par design, les politiques eBPF ne peuvent pas être supprimées pendant l'intégralité du cycle de vie de la politique, un processus ne peut pas utiliser SNAPPY pour gagner des privilèges et toute restriction de privilège est définitive, ce qui empêche toute élévation de privilèges.

— SNAPPY permet à l'administrateur de charger dans le noyau des helpers dynamiques qui peuvent être utilisés par les politiques de sécurité eBPF, permettant de réaliser des politiques de sécurité à grain fin. Si l'implémentation incorrecte par l'administrateur de helpers dynamiques peut avoir des conséquences sérieuses (déni de service, kernel panic), l'implémentation de ces codes n'est pas plus sensible que l'implémentation de modules noyau ou de code privilégié, que peut déjà réaliser l'administrateur. Par ailleurs, un administrateur malveillant a déjà la possibilité de compromettre le système simplement, sans les mécanismes de SNAPPY par exemple en supprimant le système ou en insérant un module noyau malveillant. La possibilité donnée à l'administrateur de rajouter des helpers dynamiques ne compromet donc pas plus la sécurité du système que dans une utilisation normale.

— SNAPPY met en place des limitations d'utilisation des ressources systèmes pouvant être utilisées, pour éviter qu'un processus puisse sur-utiliser SNAPPY pour créer un déni de service. Par exemple, nous mettons en place un nombre maximal de politiques eBPF et de helpers qui peuvent être ajoutés dans le système. Nous fixons aussi empiriquement une profondeur maximale de l'arbre de namespaces à 32. Ces limitations évitent qu'un processus malveillant puisse affamer le système en sur-utilisant SNAPPY.

— Finalement, il est possible de limiter quels processus peuvent mettre en place des politiques eBPF aux seuls processus qui peuvent mettre en place des politiques eBPF aux processus ayant une capabilité spécifique (`CAP_SYS_BPF`) ou même res-

treindre cette opération à l'administrateur (`CAP_SYS_ADMIN`).

Ainsi, sous les hypothèses exprimées plus haut, il n'est pas possible d'utiliser SNAPPY pour compromettre la sécurité du système ou créer des dénis de services.

3.3 Intégration avec les primitives Linux

Afin de permettre une utilisation simple de SNAPPY nous avons réalisé un certain nombre d'adaptations à des logiciels courants, qui sont présentées dans cette section.

3.3.1 Binaires usuels liés à la gestion de namespace

Afin de faciliter la création de namespace `policy_ns`, nous permettons tout d'abord l'intégration avec les binaires Linux existants de gestion de namespace. Puisque notre namespace fonctionne comme toutes les namespaces, cette intégration est triviale.

Ainsi, pour permettre au processus de changer simplement de namespace `policy_ns` en ligne de commande, nous modifions le binaire Linux `unshare`. Ainsi, nous rajoutons l'option `unshare -s` qui permet de changer de namespace SNAPPY. Pour lister les namespaces courantes, nous modifions le logiciel lsns pour qu'il puisse afficher les namespace SNAPPY.

L'utilisation de ces binaires est illustrée par le listing 3.2. Dans ce scénario, la namespace courante est initialement celle de l'hôte (soit `4026531834`). Une nouvelle namespace est alors créée, avec `unshare`. A ce stade, on trouve deux namespaces dans le système, la namespace de l'hôte et la namespace nouvellement créée. Le processus créé par unshare (pid 6151) est dans la nouvelle namespace (`4026532212`).

```
$ lsns -t snappy
NS TYPE   NPROCS     PID USER COMMAND
4026531834 snappy      3    1171 user /lib/systemd/systemd --user
$ unshare -s
$ echo $$
6151
$ lsns -t snappy
NS TYPE   NPROCS     PID USER COMMAND
4026531834 snappy      2    1171 user /lib/systemd/systemd --user
4026532212 snappy      2    6151 user -zsh
$ readlink /proc/self/ns/snappy
snappy:[4026532212]
```

Code 3.2 – Test des binaires de gestion de namespace

Ainsi, il est possible de gérer la namespace SNAPPY comme n'importe quel autre namespace Linux existantes, avec les binaires usuels Linux.

3.3.2 OCI

Il est possible de charger des politiques directement depuis une interface noyau présentée plus haut. Cette façon de charger des politiques ne permet toutefois que de charger les politiques pour un seul lancement du conteneur. Ainsi, dans le cas de conteneurs où des politiques devraient être appliquées à chaque utilisation, ce fonctionnement est insatisfaisant puisqu'il nécessiterait de recharger ces politiques à chaque lancement du conteneur. Il serait possible d'utiliser un script pour automatiser ce comportement, mais cette approche resterait *ad hoc* et peu robuste.

Afin d'appliquer des politiques à des images de conteneur, nous proposons d'intégrer SNAPPY en tant qu'extension au runtime OCI [87], la spécification d'image de conteneur la plus utilisée dans l'industrie qui est devenue le standard de facto. L'intégration au runtime OCI permet de charger les politiques OCI sans besoin d'intervention manuelle ou de script additionnel.

L'intégration au runtime OCI est réalisée grâce à l'utilisation du hook OCI `startContainer` , disponible dans la norme OCI depuis la version 1.0.0. Celui-ci permet d'exécuter des commandes depuis l'intérieur du conteneur après qu'il ait été complètement créé mais avant que la charge logicielle (entrypoint) du conteneur ne soit exécutée. Ainsi, les politiques de sécurité SNAPPY mises en place depuis ce hook sont exécutées sur toute la durée de vie du conteneur.

Le gestion des namespace `policy_ns` est réalisée à l'aide d'une extension à la spécification OCI. OCI fournit déjà un champ `"namespaces"` permettant de mettre en place les namespace du conteneur. En rajoutant un champ `policy_ns`, on est en mesure de mettre en place la namespace `policy_ns` du conteneur. Un exemple de mise en place de namespace du conteneur est montré dans le code 3.3. Dans ce code, on voit que le conteneur possède des namespaces pour toutes les namespaces Linux existantes. La namespace PID est la même que celle du processus 1234. Pour toutes les autres namespaces, y compris la namespace pour les politiques de sécurité (`"policy"`), le conteneur est chargé dans de nouvelles namespaces.

```
"namespaces": [
{
    "type": "pid",
```

```
    "path": "/proc/1234/ns/pid"
},
{  "type": "policy" },
{  "type": "network" },
{  "type": "mount" },
{  "type": "ipc" },
{  "type": "uts"},
{  "type": "user"},
{  "type": "cgroup"}
]
```

Code 3.3 – Extrait de configuration OCI lié aux namespaces de sécurité

Une fois la `policy_ns` initialisée les deux politiques SNAPPY sont chargeables dans la namespace avec le code 3.4, qui montre un exemple d'intégration à OCI. Le binaire `snappy_apply` est un programme simple responsable de charger les politiques dans le noyau. Dans cet exemple, deux politiques sont appliquées au conteneur. La première protège l'exécution de programmes (hook `BPRM_CHECK_SECURITY`). La seconde protège l'ouverture de fichiers (hook `FILE_OPEN`).

```
"startContainer" : [ {
    "path" : "/bin/snappy_apply",
    "args" : [
    "/mnt/bpf/bpf1.ko", "BPRM_CHECK_SECURITY",
    "/mnt/bpf/bpf2.ko", "FILE_OPEN",
    ]
} ]
```

Code 3.4 – Extrait de configuration OCI contenant 2 politiques de sécurité

La modification du framework runc pour à prendre en charge les politiques SNAPPY, incluant l'extension à OCI pour prendre en compte les `policy_ns` consiste 34 lignes de code. Une modification similaire appliquée au framework Docker nécessite 49 lignes de code.

Une telle intégration est par exemple très intéressante dans les cas suivants :

— Une telle configuration OCI des politiques de sécurité permet de spécifier la sécurité de l'image dans le package de l'image. Cela permet de découpler la sécurité de l'image et son contenu, facilitant l'audit et l'amélioration des politiques de sécurité.

— L'intégration à l'OCI permet aussi d'appliquer les politiques pour toute la durée de vie de l'image. Ainsi, même si l'image était initialement malveillante, les poli-

tiques de sécurité lui seraient toujours appliquées, réduisant la probabilité d'une attaque. Cette approche est donc préférable à l'approche classique consistant à d'abord lancer l'image et ensuite appliquer les politiques nécessaires à sa sécurité via l'interface noyau puisque le conteneur ne serait alors pas encore protégé dans les premières phases de son cycle de vie. L'approche consistant à d'abord créer un processus dans la nouvelle namespace, lui appliquer les politiques et lancer le conteneur seulement après n'est pas non plus satisfaisante car elle nécessite de lancer un processus additionnel dans l'hôte, ce qui n'est pas le comportement attendu.

— Dans le cadre de l'autoscaling, des images conteneurs sont régulièrement relancées pour s'adapter à la charge. L'intégration OCI proposée ici évite d'avoir à charger à chaque fois des politiques de sécurité manuellement ou à l'aide d'un script *ad hoc* qui reste insatisfaisant.

Par ailleurs, l'intégration à OCI est particulièrement intéressante car elle constitue la norme de plus bas niveau utilisée pour spécifier des images conteneurs. Il existe des spécifications de plus haut niveau comme Dockerfile ou Docker Compose. Mais puisque ces spécifications sont traduites en OCI, l'intégration à des frameworks de plus haut niveau est très simple. Nous montrons un exemple de pile logicielle basée sur OCI en figure 3.6. Puisque les logiciels sont basés en interne sur OCI, l'intégration vers ces briques de plus haut niveau est très simple. Nous montrons un exemple d'une telle intégration vers Dockerfile en section 3.3.3.

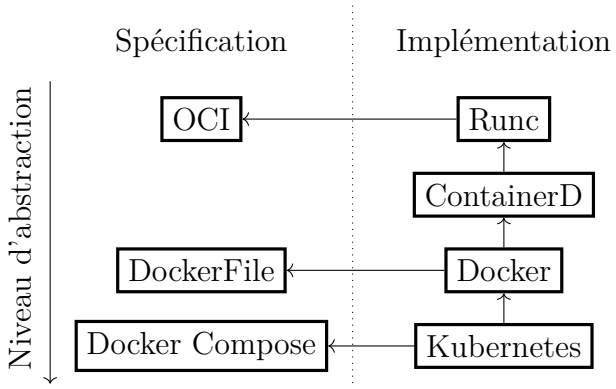


FIGURE 3.6 – Spécifications d'images conteneurs et logiciels implémentant ces spécifications

3.3.3 Dockerfile

Il est possible d'intégrer SNAPPY à des mécanismes de plus haut niveau que l'OCI comme Dockerfile. Une telle intégration implique d'ajouter une extension à la norme OCI : une commande pour mettre en place des politiques de sécurité. Cette intégration est intéressante car les conteneurs sont généralement construits avec ce type de normes de haut niveau, plus faciles à comprendre et à mettre en œuvre et moins verbeuses que les normes de bas niveau OCI. Par ailleurs, cela peut permettre de faciliter le déploiement de telles politiques dans le futur, en permettant d'appliquer de telles politiques via des hubs de conteneurs comme DockerHub.

Dans cette optique, nous proposons simplement la commande `ADDSNAPPY <policy_path> <hook_name>` qui permet de mettre en place une politique SNAPPY, utilisable dans les fichiers DockerFile. En interne, cette commande traduit la représentation des politiques en son équivalent OCI, commé présenté plus haut. Ainsi, il est possible de charger le conteneur avec les bonnes politiques intégrées au Dockerfile.

3.4 Cas d'usages

Dans cette section, nous illustrons les principaux cas d'usages de SNAPPY. Nous montrons en section 3.4.1 que SNAPPY peut être utilisé pour réaliser des politiques de sécurité permettant de réduire la surface d'attaque du serveur. Nous montrons également en section 3.4.2 que SNAPPY peut aussi être utilisé pour mitiger des vulnérabilités. Nous investiguons plus en détails ce champ d'application dans le chapitre 4 avec le framework SecuHub.

3.4.1 Réduction de surface d'attaque

Cette section montre par l'exemple la façon dont peuvent être écrites des politiques de sécurité pour réduire la surface d'attaque d'un conteneur. Par souci de clarté, nous montrons dans cette section un programme très simple. Il est à noter qu'il est possible de réaliser des programmes bien plus complexes pour bloquer tout type de comportement. Nous montrons de tels programmes dans la suite de ce document.

Prenons l'exemple d'un conteneur devant faire des calculs et ayant accès à un secret. Ce conteneur doit créer une unique connexion avec l'extérieur pour envoyer les résultats établis avec son secret. Ainsi, si ce conteneur devait créer une deuxième connexion avec

l'extérieur, il serait dans un état non prévu. Cet état pourrait être dû soit à un bogue logiciel soit à une compromission, par exemple liée à un attaquant cherchant à exfiltrer un secret. Dans tous les cas, nous ne devrions pas continuer à utiliser un tel conteneur. Une manière simple de gérer une telle situation pourrait être de reconstruire le conteneur pour repartir d'un état connu et de transmettre une alerte au Security Operations Center.

Un tel scénario d'attaque peut être mitigé simplement par une politique de sécurité SNAPPY appliquée à l'opération `socket_connect`, comme montré dans le code 3.5. La politique fonctionne comme suit : quand une connexion réseau est tentée, le hook `socket_connect` est déclenché. Nous vérifions tout d'abord si la connexion n'est pas locale en récupérant la famille de paquet et en la comparant à `AF_UNIX` à l'aide du premier helper dynamique `H_PKT`. Nous vérifions alors si la communication est la première à être tentée en ligne 4. Ceci est réalisé en regardant l'état de la namespace et en le comparant à l'état d'initialisation (0) à l'aide du helper dynamique `H_ST`. Si c'est le cas, nous incrémentons l'état de la namespace et nous pouvons accepter la connexion. Dans le cas contraire, l'état vaudra 1 et nous refusons la tentative de connexion.

```
int my_main(void* ctx) {
    if(snappy_dynamic_call(ctx, H_PKT, L_PKT_FAMILY, ctx) == AF_UNIX)
    return 0; // Local packet → OK
    if(snappy_dynamic_call(ctx, H_ST, L_ST_GET, ctx) > 0)
    return -1; // State > 0 → denied
    // We accept the first connection : we increment the counter and we allow.
    snappy_dynamic_call(ctx, H_ST, L_ST_INC, ctx);
    return 0;
}
```

Code 3.5 – Exemple de politique pour la gestion de l'unicité des connexions

Cette politique de sécurité, présentée dans le code 3.5 peut être appliquée à l'aide d'OCI à l'intégralité du cycle de vie du conteneur grâce à une configuration similaire au code 3.4, où la politique est appliquée au hook `socket_connect`. Par exemple, si la politique est compilée sous le nom `bpf_conn.ko`, nous pouvons l'ajouter à la configuration OCI en tant qu'argument à `"/bin/snappy_apply"` avec la chaine `"bpf_conn.ko", "socket_connect"`.

Le résultat de cette politique peut être vu dans le code 3.6. Comme prévu, la première connexion est acceptée. Par contre, si le conteneur tente d'établir une deuxième connexion, celle-ci sera refusée.

```
$ nc ${ip_normal} ${port} < normal_data # Normal
$ nc ${somewhere} ${port} < sensitive_data # Suspicious
```

```
(UNKNOWN) [172.17.0.2] 80 (?) : Connection refused
```

Code 3.6 – Conteneur tentant de se connecter au réseau deux fois

3.4.2 Mitigation dynamique de vulnérabilité

Dans cette section, nous montrons que SNAPPY peut être utilisé pour mitiger au niveau kernel des vulnérabilités à l'exécution sans modifier le système et avec un impact sur les performances minime. Mitiger dynamiquement des vulnérabilités à l'aide d'eBPF a récemment été décrit comme une ligne prometteuse de recherche [49]. C'est donc un cas d'usage très motivant pour SNAPPY. Ce cas d'usage est plus largement motivé et exploré dans le chapitre 4.

Dans l'optique de mitiger dynamiquement des vulnérabilités, SNAPPY peut à la fois être utilisé par l'administrateur du serveur qui souhaite protéger son infrastructure contre une vulnérabilité trouvée dans l'hôte ou un conteneur, ou par un utilisateur non privilégié visant à se protéger lui-même contre les vulnérabilités présentes dans son "domaine". Dans ce dernier cas, cela implique que les helpers dynamiques nécessaires à la mitigation de cette vulnérabilité soient déjà installés dans le système. Dans le cas contraire, il est nécessaire de passer par l'administrateur pour installer les helpers manquants. Nous en discutons en section 4.2.6 pour une application aux environnements conteneurisés.

Puisque notre mécanisme de namespaces permet d'appliquer des politiques de sécurité à des sous-ensembles du système, si une vulnérabilité peut uniquement être exploitée depuis un sous-ensemble du système, par exemple un unique conteneur, il est possible de n'appliquer la politique de sécurité qu'à cet unique conteneur grâce au mécanisme de namespaces proposé. Ceci permet d'être complètement protégé contre la vulnérabilité sans compromettre les performances du reste du système, pour lequel cette politique n'est pas appliquée.

Nous montrons cette propriété par l'exemple en proposant un correctif pour la vulnérabilité CVE-2019-5736 [79], une vulnérabilité majeure affectant le moteur de conteneurisation runc. Cette vulnérabilité possède un score CVSS v3.0 de 8.6, ce qui indique que cette vulnérabilité peut avoir des conséquences sérieuses. Cette vulnérabilité peut être exploitée pour runc jusqu'en version 1.0-rc6. Ces versions vulnérables de runc sont notamment utilisées par Docker jusqu'en version 18.09.2 et peuvent aussi être exploitées dans des logiciels comme Kubernetes, OpenShift ou LXC. Il faut toutefois noter que pour que la vulnérabilité puisse être exploitable, il faut que le conteneur soit lancé en

tant qu'administrateur. Si cette fonctionnalité est dépréciée dans beaucoup de moteurs de conteneurisation pour des raisons de sécurité, il existe encore aujourd'hui de nombreux environnements où les conteneurs restent lancés en root, et donc où une telle vulnérabilité est exploitable.

Bien que plusieurs scénarios légèrement différents existent pour cette attaque, le scénario typique d'attaque est le suivant : quand un processus est lancé à l'intérieur d'un conteneur, par exemple avec `docker exec`, le processus est créé à partir de runc en utilisant l'appel système `clone()`. Cela implique que le processus de base du conteneur est en fait le runc de l'hôte qui sera ensuite restreint pour le transformer en conteneur. Ainsi, si un conteneur construit malicieusement redirige son point d'entrée (par exemple `/bin/bash`) vers un lien symbolique pointant sur `/proc/self/exe` qui fait référence au processus courent, nous sommes en mesure d'exécuter le processus courant, c'est à dire runc *à l'intérieur* du conteneur. Il est alors possible pour le processus principal du conteneur d'attendre l'exécution de runc à l'aide d'une busy loop récupérant à l'aide de `/proc/${pid}/cmdline` le nom des programmes exécutés. Dès que runc est trouvé, le payload de l'exploit peut être exécuté : le conteneur malicieux récupère un chemin vers runc en utilisant l'appel système `open()` avec l'argument `O_PATH`. Quand l'exécution de runc est terminée, il est alors possible de réouvrir runc en écriture à l'aide de `/proc/self/fd/${fd}` bien que runc ne devrait pas être accessible depuis le conteneur. Il est alors possible d'écrire arbitrairement sur runc, par exemple pour mettre en place une version avec une backdoor ou un simple reverse shell. L'attaquant possède désormais le contrôle total du serveur en tant que root. Une description technique de cette attaque et des correctifs est également présentée dans [58].

Une manière simple de mitiger cette vulnérabilité est d'empêcher les conteneurs de réécrire sur runc. Une telle modification permet bien de mitiger une telle vulnérabilité, c'est à dire d'empêcher le conteneur malicieux de déployer leur payload, et ne bloque aucun comportement légitime puisqu'un conteneur fonctionnant normalement ne devrait *jamais* tenter de réécrire sur runc. Une telle mitigation peut être faite très simplement à l'aide de SNAPPY en ajoutant une politique de sécurité SNAPPY surveillant les ouvertures de fichier et en refusant les accès en écriture à runc. Une telle politique peut être appliquée au hook `file_open`. La mitigation de cette vulnérabilité peut être implémentée de manière naïve avec un helper spécifique dont l'implémentation est donnée en code 3.7. Cette implémentation est efficace contre la vulnérabilité mais la partie spécifique de la mitigation est faite dans le helper dynamique. Ceci n'est pas optimal puisque cela nécessite que l'ad-

ministrateur intègre dans le système un helper dynamique *spécifique* à cette vulnérabilité. Or, si SNAPPY est utilisé pour corriger un grand nombre de vulnérabilités, cela implique de mettre en place dans le système un grand nombre de helpers, augmentant la surface d'attaque de l'hôte et nécessitant que l'administrateur intervienne régulièrement. Nous présentons dans la suite de cette thèse le concept "d'helper dynamique générique" qui permet de résoudre ce problème. Nous pensons que montrer cette implémentation naïve reste intéressant car cette implémentation est très intuitive et permet d'expliciter par la suite l'intérêt des helpers dynamiques génériques. Dans ce code, nous supposons que runc est installé à l'emplacement `/sbin/runc`. La politique eBPF appelant ce helper dynamique est triviale : `return snappy_dynamic_call(helper_id, fn_id, NULL)`.

```
int dynfun(struct snappy_ctx* ctx, void** args /*unused*/) {
    struct file *f; struct dentry *d;
    f = ctx->file_ctx.file;
    d = f->f_path.dentry;
    if(!(f->f_mode & FMODE_WRITE))
        return 0; // Pas d'écriture, pas d'attaque
    if( !strcmp(d->d_iname, "runc") && // Le fichier ouvert est-il runc ?
        !strcmp(d->d_parent->d_iname, "sbin") &&
        !strcmp(d->d_parent->d_parent->d_iname, "/")) {
        pr_err("Alert: CVE-2019-5736 attempt!\n"); // Tentative d'attaque!!
        return -1;
    } else return 0; // Pas de tentative d'attaque
}
```

Code 3.7 – Helper dynamique mitigant la vulnérabilité CVE-2019-5736

Notre politique permet effectivement de bloquer la CVE-2019-5736. Dans ce code, si une telle attaque est tentée, en plus de bloquer la tentative (`return -1`), nous déclenchons une alerte à l'aide de `pr_err` permettant à l'administrateur ou au SOC de prendre les mesures nécessaires vis-à-vis des conteneurs malveillants, par exemple supprimer le conteneur, bloquer l'accès au serveur à l'attaquant, ... Nous montrerons dans le chapitre 5, consacré à l'évaluation, que la mitigation présentée ici n'impacte pas significativement les performances du système ($< 0.09\%$ dans les scénarios mesurés) et n'a aucun impact sur les performances du reste du système (ici l'hôte).

Une telle approche de patching à chaud (hotpatching) est plus simple que le correctif officiel qui a consisté à recopier runc dans un memfd, c'est à dire un fichier anonyme en mémoire découplé du binaire originel et de le réexécuter depuis ce memfd avant d'entrer

dans la namespace du conteneur pour qu'en cas de modification de runc, cela n'impacte pas le runc présent dans l'hôte. Ce correctif (195 lignes de code source pour le correctif officiel original [105]) est bien plus lourd que l'approche présentée ici. Par ailleurs, le correctif officiel augmente la taille mémoire nécessaire à l'exécution de conteneur, pouvant bloquer certains cas d'usage légitimes où des conteneurs possèdent des restrictions en mémoire [67].

3.5 Comparaison avec l'état de l'art

Le framework SNAPPY permet de mettre en place des politiques de sécurité programmables à grain fin et de niveau noyau par namespace. Dans cette section, nous le comparons à l'état de l'art.

Landlock LSM SNAPPY possède des similarités avec les anciennes versions de Landlock LSM [103], présenté en section 2.6.1, puisque ce dernier permet également aux processus non privilégiés, y compris conteneurisés, de mettre en place des politiques de sécurité pour se protéger eux-mêmes des attaques. Comme pour Landlock, les politiques SNAPPY ne sont pas désactivables, sont empilables et peuvent être appliquées à des sous-ensembles du système. Toutefois, contrairement à Landlock, les politiques SNAPPY peuvent être appliquées à tous les hooks LSM (à travers une couche d'abstraction) et pas seulement à quelques hooks liés à la gestion des fichiers, du ptracing et des credentials comme c'est le cas avec Landlock. SNAPPY utilise une namespace dédiée, ce qui lui permet notamment de gérer la virtualisation imbriquée sans risque d'élévation de privilèges, y compris quand un conteneur est créé depuis un démon système. Ceci n'est pas possible avec le système de hiérarchie de processus de Landlock. Enfin, le concept d'helper dynamique introduit dans cette thèse permet d'avoir des politiques plus flexibles qu'avec de l'eBPF classique, utilisé par Landlock LSM.

Security Namespaces Nous avons présenté les Security Namespaces [115] en section 2.7.1. Security Namespaces permet de mettre en place des politiques de sécurité namespacées pour plusieurs LSMs existants. Nous montrons que les limitations de ce framework sont levées dans SNAPPY. SNAPPY permet de mettre en place des politiques pour tout code eBPF valide et peut se servir des helpers dynamiques pour mettre en place des politiques très flexibles. Cela permet de mettre en place des politiques beaucoup plus variées que celles, limitées, pouvant être mises en place par un unique module LSM, comme c'est

le cas dans Security Namespaces. De plus, SNAPPY se base sur des primitives connues (eBPF, LSM, namespaces), il est donc plus simple de vérifier que ce framework fonctionne correctement que pour Security Namesapaces, qui requiert une adaptation *manuelle* des modules LSM adaptés. Enfin, le surcoût en performances engendré par SNAPPY est plus faible que celui de Security Namespaces. Ceci est principalement dû au fait que SNAPPY requiert de vérifier la `policy_ns` active et ses ancêtres (typiquement moins de 3), et pas *toutes* les namespaces comme pour Security Namespaces. Ainsi, nous pensons que l'approche SNAPPY est préférable pour mettre en place des politiques pour les conteneurs non privilégiés, à celle proposée par Security Namespaces.

BPFBox et BPFContain Les framework BPFBox [36] et BPFContain [35], présentés en section 2.6.3, permettent de mettre en place des politiques eBPF à divers points du noyau au niveau des hooks LSM. Grâce à BPFContain, il est possible d'utiliser ces politiques pour protéger des conteneurs. Toutefois, ces politiques devant être utilisées par un administrateur, BPFContain ne peut pas être utilisé par un conteneur non privilégié pour se protéger lui-même comme c'est possible avec SNAPPY. De plus, BPFContain hérite aussi des limitations d'eBPF, levées par les helpers dynamiques proposés par SNAPPY. Enfin, ce framework ne dispose pas de namespace et souffre donc des limitations associées.

Falco Falco [101] permet également de mettre en place des politiques eBPF pour protéger des conteneurs. Toutefois, ce framework ne permet que des politiques pour filtrer par appels système et pas par hooks LSM, ce qui est une approche plus complexe d'utilisation et moins unifiée. Par ailleurs, Falco ne contient pas d'espace de nommage et souffre donc des limitations associées. Enfin, l'utilisation de Falco nécessite les droits administrateurs et ne peut donc pas être utilisé par un conteneur non privilégié.

KRSI le framework KRSI [101], présenté en section 2.6.2 permet également de mettre en place des politiques pour les hooks LSM. Ce framework possède les mêmes limitations que BPFBox et BPFContain. Par ailleurs, les politiques appliquées via ce framework s'appliquent à tout le système. KRSI ne peut donc pas être utilisé pour protéger individuellement un conteneur.

Seccomp-BPF Seccomp-bpf [46] permet de mettre des politiques eBPF pour protéger des appels systèmes qui est une approche moins flexible que le filtrage au niveau LSM.

Par ailleurs puisque seccomp-BPF ne permet pas de déréférencer les pointeurs noyau, les politiques pouvant être exprimées par ce framework sont très limitées.

Approches spécifiques Certaines approches permettent de vérifier des propriétés spécifiques d'un conteneur, comme leur intégrité ou la validité du trafic réseau. Ces approches ne constituent donc pas une alternative réelle aux politiques de sécurité flexibles de SNAPPY. Les frameworks DIVE, CloudVaults, IPE, Cilium et BASTION relèvent de cette catégorie.

Pour récapituler la comparaison entre SNAPPY et l'état de l'art, nous reprenons la table 2.3 et y ajoutons le framework SNAPPY dans la table 3.2. SNAPPY est le seul framework validant tous les indicateurs techniques mesurés dans ce tableau. SNAPPY est donc une solution avantageuse par rapport à l'existant pour la mise en place de politiques de sécurité. En travaillant sur SNAPPY, il pourrait être possible d'améliorer sa maturité et transformer ce projet en une solution pratique pour améliorer la sécurité système et conteneur en production.

3.6 Limites de SNAPPY

Les principales limitations de SNAPPY dans sa forme actuelle sont :

— Dans l'implémentation actuelle, la cohérence des helpers n'est pas vérifiée. Cette limitation est acceptable au sens où dans les hypothèses réalisées, on admet une confiance en l'administrateur de l'hôte. Toutefois, rajouter des vérifications de cohérence des helpers pourrait limiter le risque qu'un bogue soit introduit à cause d'une mauvaise implémentation d'helpers.

— Un helper ne doit pas pouvoir appeler le hook qui l'a appelé. Cela causerait une récursion infinie. Il faut toutefois noter que ce comportement n'est pas censé arriver en pratique. En effet, une implémentation correcte de code noyau doit toujours utiliser les fonctions internes, et pas les wrappers qui pourraient déclencher des hooks LSM à nouveau. Cette limitation ne doit donc pas poser de problème majeur en pratique. Il est toutefois possible de mettre en place des "garde-fous" en interdisant l'appel à de telles fonctions.

— Actuellement, les helpers sont identifiés par des numéros. S'il est possible d'abstraire ces numéros à l'aide de macros, cette identification reste manuelle. L'ajout d'un mécanisme permettant de voir quels helpers et politiques sont actuellement

	Basé sur : Config(C), Règles(R) ou Code(X)	Champ d'action	Extensibilité	Adaptation aux conteneurs	Utilisation interactive non privilégiée	Maturité
SNAPPY	**X**	●	●	●	●	○
Outils de configuration	C	●	○	●	○	●
DIVE [29]	C	○	○	●	○	○
CloudVaults [65]	C	○	○	●	○	○
IPE [74]	C	○	○	○	○	○
Cilium [21]	C	◐	○	●	○	●
Security Namespaces [115]	R	◐	○	●	○	○
BASTION [76]	R	○	○	●	○	○
Landlock v>14 [102]	R	◐	○	●	●	●
Landlock v<14 [103]	X	●	◐	●	●	○
eBPF noyau [61]	X	●	◐	○	○	●
KRSI [110]	X	●	◐	○	○	●
BPFBox + BPFContain [36, 35]	X	●	◐	●	○	○
SeccompBPF [46]	X	●	○/◐	●	●	●
Falco [101]	X	●	◐	●	○	●

TABLE 3.2 – Comparaison de SNAPPY avec l'état de l'art

ajoutés dans le système dans un format humainement lisible permettrait d'éviter ce problème. Le mécanisme d'identification de politiques présenté dans le chapitre suivant va dans cette direction.

Dans le chapitre suivant, nous présentons le design et l'implémentation de notre contribution SecuHub, un cas d'usage de SNAPPY permettant la distribution et l'installation simple de politiques de mitigations pour CVEs, notamment applicable aux conteneurs.

Chapitre 4

SecuHub

Résumé du chapitre *Dans ce chapitre, nous étudions en détails un cas d'usage prometteur du framework SNAPPY : la mitigation de CVEs pour conteneurs avec notre contribution SecuHub, une solution de distribution de bout-en-bout et de gestion de politiques de sécurité pour les conteneurs avec pour visée particulière la mitigation individuelle de vulnérabilités CVE données. Nous détaillons les principes de distribution unifiée et d'installation sécurisée des politiques de mitigation permettant une exécution machine-indépendante de celles-ci. Nous illustrons les capacités de mitigation de SecuHub avec l'étude de politiques de mitigation de CVEs réelles et nous comparons SecuHub avec l'état de l'art.*

Contents

4.1 Motivation et objectifs

4.1.1 Contexte

L'approche la plus classique pour corriger les vulnérabilités en environnement conteneur est d'utiliser un correctif logiciel dès que celles-ci sont trouvées. En effet, en corrigeant les logiciels rapidement, aucune vulnérabilité connue n'est exploitable dans le logiciel à un instant T. La réalisation d'une attaque implique alors d'exploiter une faille 0-day, ce qui est bien moins probable en environnement réel que l'exploitation d'une vulnérabilité connue. Toutefois, nous avons expliqué en chapitre 2 section 2.1.3 que même en cas de disponibilité d'un correctif logiciel, la vulnérabilité restait dangereuse. Pour ces raisons, un correctif logiciel n'est pas toujours une approche suffisante pour mitiger des vulnérabilités.

Ainsi, nous avons vu dans le chapitre 2 que parallèlement aux correctifs, il est possible de mitiger la vulnérabilité. Nous avons par ailleurs montré que les approches basées sur le code permettaient de faire du filtrage à grain plus fin que les autres approches et que si les approches basées sur le code étaient largement explorées en recherche, elles restaient assez peu utilisées en production. Nous pensons que fournir des moyens permettant de mettre en place très simplement des politiques codées faciliterait leur utilisation et pourrait permettre un usage plus massif la mitigation de vulnérabilités. Ceci permettrait de gagner à la fois en termes de sécurité en permettant de simplement d'éliminer les vulnérabilités connues des images logicielles et en termes de performances en exécutant les politiques de sécurité plus efficacement qu'avec des systèmes de règles et de configuration.

Les primitives que nous présentons dans ce chapitre sont génériques et utilisables pour protéger toute `policy_ns`. Toutefois, de par l'intérêt du cas d'usage des conteneurs, nous traitons principalement le cas des conteneurs dans ce chapitre.

Au meilleur de nos connaissances, il n'existe pas aujourd'hui de solution permettant une distribution de bout-en-bout et une gestion de politiques de sécurité noyau à grain fin pour les conteneurs, avec la visée particulière de mitiger individuellement des failles de sécurité qui les affectent. Nous proposons dans cette thèse Secuhub, une plateforme de distribution et de mise en place de politiques de mitigation avec pour objectif de filtrer au niveau noyau l'exploitation de CVE particulières. Nos politiques sont identifiées grâce à l'identifiant de CVE de l'attaque qu'elles mitigent.

4.1.2 L'approche SecuHub

Le cas d'usage de la distribution de politiques de mitigation noyau applicables aux conteneurs présenté dans ce chapitre, est à notre avis très motivant et tire particulièrement avantage des capacités de SNAPPY pour résoudre un problème important. Nous proposons la plateforme de distribution SecuHub qui permet aux utilisateurs de conteneurs de télécharger et d'installer de manière simple et sécurisée des politiques de mitigation à grain fin permettant de mitiger des vulnérabilités données. Nos politiques de mitigation sont identifiées par le numéro de la vulnérabilité (CVE) qu'elles mitigent et visent à bloquer le vecteur d'attaque sans bloquer aucun comportement légitime. SecuHub résout les problèmes techniques liés à la distribution unifiée et l'installation sécurisée de politiques de mitigation pour permettre la distribution et l'exécution transparente pour l'utilisateur de politiques de mitigation. Comme expliqué dans la section suivante, cela implique de résoudre le fait que ces politiques de sécurité écrites en eBPF, font référence à des helpers dynamiques selon une convention machine-dépendante. Un mécanisme adapté est donc nécessaire pour permettre la distribution et l'installation de telles politiques indépendamment de la machine sur laquelle elles sont déployées.

Pour résumer, nous proposons avec SecuHub un changement de paradigme. SecuHub permet de télécharger et d'installer des politiques de sécurité aussi simplement que nous téléchargeons aujourd'hui des applications à travers des magasins d'applications (App Stores). Avec l'approche SecuHub, il suffit de demander une politique de mitigation identifiée par l'identifiant de CVE qu'elle mitige pour l'installer. De cette manière nous découplons les efforts de mitigation d'une application vulnérable de la mise en place d'un correctif. Le développeur de l'application vulnérable n'est alors plus le seul en mesure de gérer une vulnérabilité nouvellement trouvée : un tiers de confiance est également en mesure de développer une politique de mitigation qui bloque le vecteur d'attaque. Ce tiers de confiance met une telle mitigation à disposition des utilisateurs du logiciel vulnérable via un "hub" logiciel, SecuHub. Ce tiers de confiance n'a pas besoin d'accéder au code du logiciel vulnérable. La politique mitige alors totalement la vulnérabilité dès qu'elle est chargée dans le noyau. L'application de telles politiques ne nécessite pas de redémarrer le logiciel vulnérable.

Nous montrons dans cette thèse des exemples motivants d'utilisation de politiques de mitigation qui couvrent des CVE récentes. Nous montrons également qu'en pratique, nous sommes en mesure de créer des helpers dynamiques spécialisés par exemple pour la gestion des chaines de caractères, la gestion de l'accès aux fichiers, ... Nous montrons de manière

empirique qu'il est possible de gérer un très grand nombre de cas avec un nombre restreint de helpers, augmentant ainsi la versatilité de SecuHub et limitant son impact en mémoire. Puisqu'il est possible de n'utiliser qu'un nombre restreint de helpers pour effectuer des traitements complexes, ces helpers peuvent être aisément audités afin de garantir leur sécurité. Nous ne prétendons pas avoir étudié exhaustivement la taxonomie complète des vulnérabilités pouvant être mitigées à l'aide de SecuHub et celles qui tombent hors de son champ d'application mais nous présentons un certain nombre de cas d'usages qui éclairent la généralisation d'une telle approche.

Nous pensons qu'en couplant SecuHub à des logiciels de détection de vulnérabilités, comme Trivy[6], Vuls[63], Clair [97] ou Grype [5], qui sont en mesure de répertorier les logiciels installés et de trouver les CVEs présentes/exploitables dans un logiciel/conteneur, il devrait être possible de (semi-)automatiser la mitigation de vulnérabilités. Bien que ce couplage soit hors du domaine de cette thèse, nous pensons que SecuHub pourrait faciliter le déploiement de politiques de mitigation des vulnérabilités identifiées, permettant de laisser le conteneur sûr même lorsqu'une nouvelle vulnérabilité est trouvée.

4.1.3 Contributions

Dans ce chapitre, nous présentons les contributions suivantes :

— Nous présentons le design et l'implémentation de SecuHub qui permet de déployer et d'installer dans le noyau des politiques de mitigation applicables par conteneur.

— Nous présentons un certain nombre de politiques de mitigation qui bloquent précisément des CVEs rééelles sans bloquer de comportements légitimes et avec un impact minimal sur les performances.

— Nous montrons que notre concept de helper dynamique est viable : il est possible de distribuer et d'installer dynamiquement et très simplement des helpers dynamiques utilisables par les politiques de mitigation, permettant de mitiger un grand nombre de vulnérabilités sur une pluralité d'hôtes.

Travaux de l'auteur sur cette thématique

— SecuHub : Distributing Kernel-level Security Policies for Container Vulnerabilities Mitigation. **Maxime Bélair**, Sylvie Laniepce, and Jean-Marc Menaud. En cours de soumission.

— Procédé et module d'installation d'un programme de mitigation dans le noyau d'un équipement informatique. **Maxime Bélair**, Sylvie Laniepce, N° dépôt : 20 2314.

Déposé le 2/03/2021.

Organisation du chapitre Ce chapitre est organisé comme suit : après une présentation du design global de SecuHub en section 4.2.1, nous présentons le processus d'installation des politiques de securité SecuHub en 4.2.2, l'installation des helpers dynamiques en 4.2.3. Nous présentons ensuite la manière dont l'édition des liens est réalisée dans Secuhub en section 4.2.4, l'authentification des politiques en section 4.2.5 et le design des helpers dynamiques génériques en section 4.2.6. Nous illustrons le fonctionnement de SecuHub avec la correction de plusieurs failles récentes en section 4.3. Nous terminons ce chapitre en comparant l'approche SecuHub à l'état de l'art en section 4.4 et en présentant les limites de Secuhub en section 4.5.

4.2 SecuHub : Design et implémentation

4.2.1 Design global

SecuHub consiste en une architecture client-serveur qui fournit un moyen simple de stocker des politiques de mitigation dans un hub centralisé, que nous appelons ici "serveur SecuHub", et de les distribuer à la demande au "client SecuHub". Tous les processus sont en mesure d'utiliser SecuHub du moment que SecuHub client et les interfaces SNAPPY leur sont mis à disposition. Cette mise à disposition est faite en montant (`mount`) automatiquement le client Secuhub et les interfaces noyau correspondantes installées dans l'hôte, dans les conteneurs. Les conteneurs sont alors en mesure d'utiliser SecuHub. Dans l'implémentation actuelle, il est également nécessaire qu'une chaine de compilation eBPF (llvm) soit installée et mise à disposition du client SecuHub pour pouvoir compiler les politiques téléchargées depuis le serveur SecuHub.

Une vue d'ensemble de l'architecture SecuHub est donnée par la figure 4.1. Le client SecuHub est un script exécuté en espace utilisateur qui peut être soit exécuté par un conteneur pour se protéger lui-même, soit par l'administrateur pour protéger des hiérarchies de processus ou pour se protéger lui-même contre des processus. SecuHub Client est responsable du téléchargement des politiques depuis le serveur SecuHub. Une fois les politiques de mitigation téléchargées, elles sont compilées par le client SecuHub, éventuellement après des configurations. Le client SecuHub utilise alors l'interface noyau SNAPPY pour charger ces politiques dans le noyau. Ces politiques de mitigation sont mises en place dans

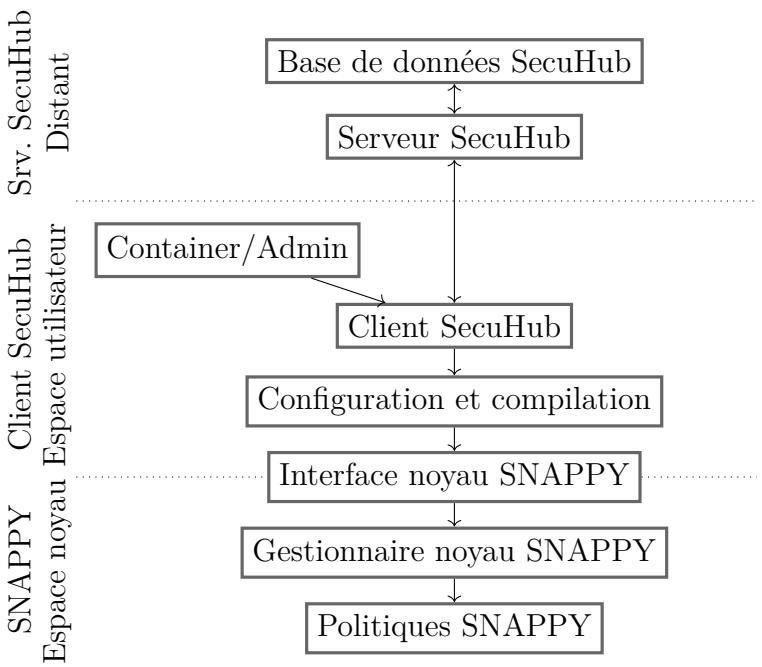


FIGURE 4.1 – Design global de SecuHub

le système à l'aide du gestionnaire noyau SNAPPY. La politique est finalement chargée dans la `policy_ns` du conteneur à protéger et devient immédiatement appliquée.

Puisque SecuHub est basé sur SNAPPY, et que le client SecuHub est monté par défaut dans les conteneurs, tous les processus sont en mesure d'utiliser le client SecuHub pour charger des politiques de mitigation dans leur *propre* `policy_ns`. Il est également possible pour un processus de charger une politique de mitigation dans une namespace fille de la namespace courante. Par contre, il n'est pas possible de charger des politiques de sécurité pour une namespace qui n'appartient pas à la descendance de la namespace courante. Cette limitation empêche que des processus malveillants soient en mesure de réaliser des DoS sur le système en appliquant des politiques sur des namespaces pour lesquelles ils ne sont pas en contrôle. Par ailleurs, quand une politique est appliquée à une namespace, elle est aussi appliquée à sa descendance, le cas échéant. Ainsi, créer une nouvelle namespace ne permet pas de gagner de privilège. Enfin, quand une politique de mitigation est chargée dans le système, il n'est pas possible de la supprimer durant l'intégralité du cycle de vie de la namespace pour éviter tout gain de privilège. Grâce aux limitations décrites dans ce paragraphe, il est possible pour des namespaces d'augmenter la protection du système de par l'installation de politique de mitigation de vulnérabilité et ces protections sont

définitives. Si une namespace est protégée contre une vulnérabilité à un instant donné, cette namespace et son éventuelle descendance resteront protégées durant tout leur cycle de vie.

Nous rappelons par ailleurs que comme pour SNAPPY, il est possible de configurer les processus en capacité de charger des politiques dans leur propre namespace à 1) l'intégralité des processus, 2) ceux disposant d'une capabilité spécifique `CAP_BPF`, ou 3) uniquement l'administrateur système `CAP_SYS_ADMIN` qui peut alors mettre en place les politiques pour d'autres processus. Il faut toutefois noter que contrairement à SNAPPY, les politiques de mitigations chargées dans le cadre de SecuHub sont des politiques provenant d'une source de confiance, devant être auditées et pouvant être vérifiées à tout moment, et en aucun cas du code arbitraire eBPF comme c'est le cas avec SNAPPY. Il est possible de configurer SNAPPY pour qu'il n'accepte de charger que de telles politiques authentifiées en provenance du serveur SecuHub. Cette configuration est faite au niveau SNAPPY et pas SecuHub pour qu'elle ne soit pas contournable via l'utilisation directe des interfaces noyau de chargement de politiques. Dans ce contexte, il ne nous semble pas nécessaire de restreindre les utilisateurs pouvant charger des politiques de sécurité. La probabilité qu'une éventuelle faille du vérifieur eBPF puisse être exploitée en utilisant uniquement du code venant d'une source de confiance visant à mitiger une vulnérabilité, semble faible voire négligeable.

Le serveur SecuHub a été implémenté en tant que serveur PHP (environ 450 lignes de code) et le client SecuHub a été implémenté comme un script Bash (environ 950 lignes de code). Le client SecuHub s'interface en couplage faible à la fois avec les interfaces noyau SNAPPY et le serveur SecuHub, pour implémenter le protocole de distribution de politiques de mitigation, présenté dans la section suivante. Le chargement et l'exécution des politiques de sécurité sont gérés par SNAPPY, tels que décrits en chapitre 3.

4.2.2 Installation de politiques de mitigation

La procédure d'installation de politiques de mitigation est illustrée par la figure 4.2. Elle montre comment le client SecuHub, le serveur SecuHub et SNAPPY coopèrent pour installer des politiques de mitigation. D'un point de vue utilisateur, il est possible de réaliser toute cette procédure en une seule ligne de commande. Dans l'énumération suivante, les numéros de paragraphe correspondent aux étapes de chargement indiquées sur la figure 4.2.

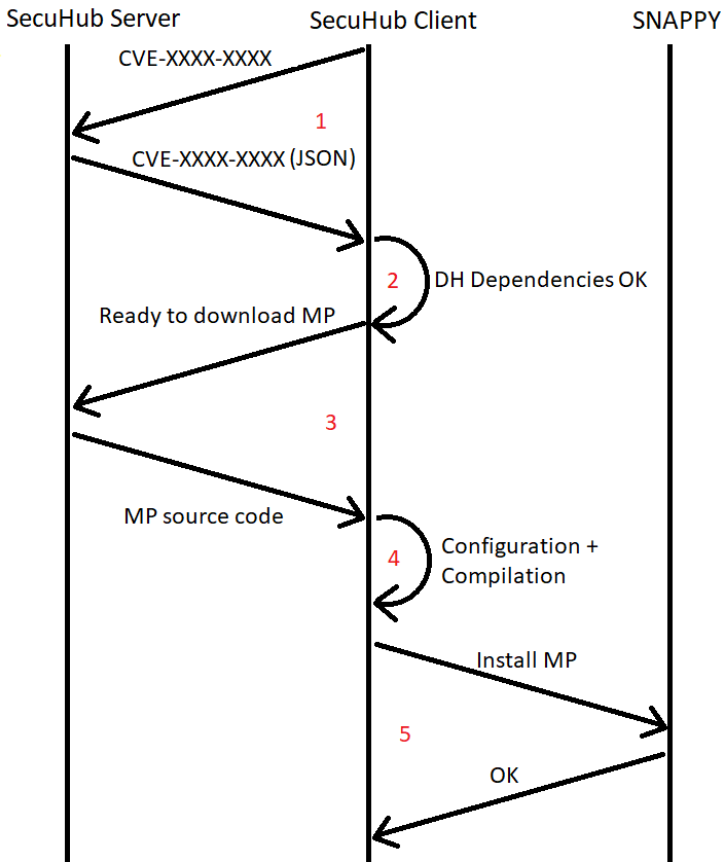


FIGURE 4.2 – Installation de politiques de mitigation SecuHub

1. Une entité (conteneur, administrateur, ...) utilise le client SecuHub pour mettre en place une politique de mitigation avec la commande `secuhub --install-ebpf CVE-XXXX-XXXX` où `CVE-XXXX-XXXX` identifie la vulnérabilité à mitiger. En retour, le serveur renvoie un fichier de spécification JSON indiquant toutes les données nécessaires au chargement dans le système de cette politique mitigeant la CVE. Ce fichier contient :
   — La liste des helpers dynamiques nécessaires, identifiés par : leur nom unique (par exemple `STRING_HELPER`), leur valeur de hash et la liste des fonctions support qu'ils fournissent.
   — Les attributs de la politique de mitigation à charger, à savoir le nom unique du code, son hash, les hooks LSM sur lesquels mettre en place la politique de mitigation et si besoin la liste des variables de configuration qui doivent être renseignées pour s'adapter à l'environnement de l'hôte.

2. Afin de vérifier que la politique de mitigation peut effectivement s'exécuter dans l'hôte, le client SecuHub vérifie si les helpers dynamiques nécessaires à l'exécution de cette politique sont bien présents sur l'hôte. Puisque par design SecuHub utilise des helpers génériques de telle manière à ce qu'un petit nombre de helpers soit en mesure de gérer la majorité des cas, les helpers requis sont généralement déjà présents dans le système. Toutefois, si un helper dynamique venait quand même à manquer, le processus d'installation de la politique de mitigation s'interrompt et ce helper doit être installé par l'administrateur, selon le processus décrit en section 4.2.3. L'installation de la politique peut alors reprendre.
3. Le code source des politiques de mitigation (eBPF) est téléchargé depuis le serveur SecuHub.
4. La politique de mitigation est compilée localement sur l'hôte pour le transformer en objet kernel, exécutable. L'édition des liens vers les helpers dynamiques est alors réalisée, cette procédure est détaillée en section 4.2.4. Si besoin, les variables de configuration sont mises en place à l'aide de l'option de compilation `-D` pour adapter la politique à l'environnement d'installation.
5. Dès que la politique de mitigation est chargée dans le noyau via les mécanismes de SNAPPY, elle devient en utilisation. La vulnérabilité est à présent mitigée.

4.2.3 Installation d'helpers dynamiques

Dans certains cas, des helpers peuvent être manquants dans le système. Dans le cas où un helper dynamique est absent du système, celui-ci doit donc être installé par l'administrateur pour permettre l'installation de politiques de mitigations nécessitant ce helper. Cette installation se fait avec le client SecuHub et possède de grandes similarités avec la procédure d'installation des politiques de mitigation, comme détaillé ci-dessous :

— L'administrateur exécute la commande `secuhub --install-helper <HELPER_NAME>` où `<HELPER_NAME>` correspond au nom du helper dynamique à installer, ce nom est notamment indiqué dans le fichier de spécification JSON de la politique de mitigation qui précise les helpers requis. En retour, le serveur SecuHub renvoie au client un fichier JSON spécifiant le doublet (`<HELPER_NAME>`, `<HELPER_HASH>`) ainsi que la liste des fonctions support fournies par le helper dynamique. Dans l'implémentation actuelle, le code du helper dynamique est alors téléchargé en tant que code objet où l'édition des liens n'a pas encore été réalisée.

— L'édition des liens est alors réalisée à l'aide de la fonctionnalité `defsym symbol=value` de `ld`. Le KASLR (Kernel Address Space Layout Randomization) [24] est géré automatiquement en récupérant l'offset noyau, en comparant l'adresse réelle et l'adresse sans KASLR, d'un symbole connu `_text`, dont l'adresse est codée en dur. Cet offset est retiré de la table globale d'offset du binaire. Ainsi, quand le code est chargé, il est possible de rajouter le bon offset KASLR pour réaligner les appels de fonctions aux adresses des symboles noyau.

— SNAPPY met alors en place le helper dynamique dans le noyau avec les métadonnées contenant les valeurs de `<HELPER_NAME>` et `<HELPER_HASH>` récupérées en même temps que `<KERNEL_ID>` qui est un identifiant de helper dynamique incrémental attribué par SNAPPY lors de chaque installation. Le doublet (`<HELPER_NAME>`, `<HELPER_HASH>`) est alors associé à `<KERNEL_ID>` l'identifiant machine-dépendant de helper dynamique utilisé par SNAPPY.

— Le helper dynamique est alors utilisable de manière transparente par les politiques de mitigation SecuHub. Le nouvel helper peut dès à présent être utilisé par les politiques eBPF en utilisant la convention d'appel SecuHub. Les politiques pour lesquelles ce helper était manquant peuvent maintenant être installées.

4.2.4 Édition des liens des politiques de mitigation

Comme indiqué précédemment, il existe deux conventions d'appel pour les helpers dynamiques : celle utilisée par les politiques de mitigiation stockées dans le serveur distant SecuHub (machine-indépendante) et celle utilisée par SNAPPY (machine dépendante). Ainsi, les politiques de mitigation qui utilisent la convention d'appel `secuhub_helper(<HELPER_NAME>, <HELPER_HASH>, <fn_id>, <args>)` doivent être linkées vers la convention d'appel de SNAPPY `snappy_dynamic(<KERNEL_ID>, <fn_id>, <args>)` pour que ce dernier soit en mesure de les exécuter.

Cette transposition peut être réalisée de manière très simple grâce à l'association entre le doublet (`<HELPER_NAME>`, `<HELPER_HASH>`) et `<KERNEL_ID>`, construite lors de l'installation des helpers dynamiques comme décrit dans la section précédente.

En pratique, pour faire ce linking de manière automatique, un header C est utilisé, automatiquement inclus à la compilation. Ce header contient la liste des helpers disponibles dans le système dans le format `#define __SECUHUB_<HELPER_NAME>_<HELPER_HASH>__<KERNEL_ID>`, maintenant ainsi le lien entre le doublet (`<HELPER_NAME>`, `<HELPER_HASH>`) et le `<KERNEL_ID>`. Cette transposition est donc réalisée très simplement en utilisant une

X-macro, comme indiqué dans le code 4.1.

```
#define secuhub_helper(name, hash, fn_id, args) \
    snappy_dynamic(__SNAPPY_ID_ ## name ## _ ## hash ## __, fn_id, args)
```

Code 4.1 – Transposition vers la convention d'appel SNAPPY

4.2.5 Authentification des politiques

SecuHub permet de distribuer des politiques de mitigation de manière sécurisée. Il est donc nécessaire de mettre en place des mécanismes d'authentification pour s'assurer que les politiques installées par SNAPPY proviennent effectivement de SecuHub. Nous avons déjà expliqué que le client SecuHub était monté en lecture seule dans les conteneurs depuis l'hôte. Puisqu'il est nécessaire d'avoir confiance en l'hôte pour lancer des conteneurs sur ce dernier, il est de fait possible de considérer également le client SecuHub comme de confiance dans ce scénario, ce dernier appartenant également à l'hôte. Nous montrons dans cette section que des mécanismes simples d'authentification suffisent pour avoir une utilisation sécurisée de SecuHub dans le cadre d'une utilisation en environnement conteneurs.

— Les politiques proviennent du serveur SecuHub, de confiance
— Les politiques sont téléchargées par le client SecuHub également de confiance. Le client SecuHub est en charge d'authentifier les politiques téléchargées pour s'assurer que ces dernières n'ont pas été compromises (par exemple via une attaque de l'homme du milieu).
— Les seules modifications opérées sur les politiques reçues de SecuHub serveur sont réalisées réalisées par le client SecuHub de confiance. Il s'agit de la compilation de politiques. Cela ne pose donc pas de problème de sécurité.
— Les politiques sont chargées depuis le client SecuHub de confiance vers le noyau, également de confiance.
— L'utilisateur est à tout moment en mesure de vérifier les politiques chargées dans sa namespace à l'aide d'une interface noyau spécifique : `/sys/kernel/security/snappy/current_policies`.

Ainsi, il est possible d'utiliser des mécanismes de cryptographie à clef publique standard pour que les politiques installées puissent être signées par SecuHub Serveur et que leur signature puisse être vérifiée par SecuHub client. Les questions cryptographiques additionnelles qui peuvent se poser comme la révocation de certificat peuvent être ré-

solus avec des mécanismes cryptographiques existants. Puisque ces mécanismes peuvent être intégrés en couplage faible avec l'implémentation de SNAPPY et SecuHub, nous les considérons en dehors du champ d'intérêt de cette thèse.

Si en environnement conteneurs, le fait de considérer le client SecuHub comme de confiance ne pose pas de problème puisque ce dernier est monté depuis l'hôte, ce n'est pas le cas dans tous les environnements. Nous discutons des possibilités d'utilisations de SecuHub tout en n'ayant pas confiance en le client SecuHub en section 6.2.2.

4.2.6 Helpers génériques

4.2.6.1 Objectif

Dans cette section, nous montrons de quelle manière il est possible d'écrire des helpers génériques pour supporter des politiques de mitigation de CVEs. Nous montrons grâce à l'exemple illustratif de `STRING_HELPER` que de tels helpers dynamiques peuvent effectivement être génériques et donc réutilisables pour supporter la mitigation de nombreuses vulnérabilités. Ainsi, nous pensons qu'un petit nombre de helpers dynamiques doit être en mesure de mitiger la grande majorité des CVEs.

4.2.6.2 Design

L'objectif des helpers génériques est de permettre la gestion de toutes les opérations couramment nécessaires pour une famille de cas d'usages, par exemple la gestion des fichiers, des chaines de caractères, du ptracing, ... Chaque helper est responsable d'une famille de cas d'usages et cherche à la traiter le plus génériquement possible. Nous visons à ce que le helper se suffise à lui-même dans son domaine de compétences.

Afin d'atteindre cet objectif, les helpers dynamiques possèdent deux niveaux de flexibilité. Les helpers doivent pouvoir gérer un grand nombre de traitements sur un grand nombre de données d'une même famille. Ce design s'oppose au design des helpers statiques où les fonctions proposent un traitement spécifique sur une donnée spécifique. Par exemple, `bpf_get_current_pid_tgid` et `bpf_get_current_uid_gid` récupèrent respectivement le (pid, tgid) et le (uid, gid) du processus courant. Au contraire, dans un helper générique une seule fonction support est en mesure de récupérer toutes les valeurs liées au thread, la valeur à récupérer étant passée en argument sous forme de constante. La manière dont peut être implémentée une telle généricité est explicitée dans la prochaine sous-section.

4.2.6.3 Exemple : `STRING_HELPER`

Nous montrons comment peuvent être implémentés des helpers dynamiques génériques par l'exemple en présentant `STRING_HELPER`, le helper qui gère les chaines de caractères. Les politiques doivent souvent avoir accès à des chaines de caractères et être en mesure de faire des traitements sur celles-ci afin de prendre des décisions de sécurité. Par exemple, une politique de mitigation attachée au hook `BPRM_CHECK_SECURITY`, déclenché lors du lancement d'un processus, peut nécessiter de réaliser des vérifications de sécurité sur plusieurs chaines de caractères : le chemin du binaire exécuté, les arguments du programme, les variables d'environnement, ... Les traitements réalisables sur ces chaines de caractères peuvent être de comparer une chaine de caractère à une (des) valeurs interdites, récupérer la valeur d'un token dans un texte délimité par des délimiteurs spécifiques ou réaliser une glob/regex. Conformément à l'objectif des helpers dynamiques, toute cette famille de traitements peut s'appliquer sur des objets différents mais similaires c'est à dire des chaines de caractères et des tableaux de chaines de caractères, dans `STRING_HELPER`.

```
int function_example(struct snappy_ctx* ctx, void** args) {
    if(!valid_input(args, EXPECTED_ARGS_NUMBER))
    return -EINVAL;
    char** arr = getStringArr(ctx,*((int*)args[ARG_ID_OFF]));
    int str_id = *(int*)(args[STR_ID_OFF]);

    // For string arrays, we can perform checkings on single values and arrays
    // If arg_id > 0 we make the handling on the corresponding value
    // If arg_id < 0 we make the handling on [arg_id-1; array_size[
    do {
        char* str = arr[str_id >= 0 ? str_id : -str_id];
        /* e.g. if(!strncmp(...)) return true; */
        ret = perform_treatment(...);
        if(ret == TO_FIND)
        return ret;
    } while(arg_id<0 && arg_id > -bprm_ctx.bprm->argc);
    return 0;
}
```

Code 4.2 – Design du helper `STRING_HELPER` (légèrement simplifié)

Toutes les fonctions support disponibles dans `STRING_HELPER` suivent le même design, comme montré dans le code 4.2.

— La fonction `valid_input` vérifie que les entrées sont fournies dans le bon format.

Dans le cas contraire, le helper retourne un code d'erreur et ne réalise aucune opération.

— La fonction `getStringArr` récupère par la suite la chaine ou le tableau de chaines de caractères sur lesquels les traitements doivent être réalisés.

— Si on travaille sur un tableau de chaines de caractères, on récupère les élément(s) sur lesquel(s) doit être fait le traitement. Si on travaille sur une chaine de caractères simple, la valeur de `str_id` doit valoir 0 ou la fonction `valid_input` aura déjà retourné un code d'erreur.

— Dans le corps de la boucle, on récupère la chaine sur laquelle faire le traitement et on l'effectue. Par exemple, si le traitement à effectuer est une comparaison de chaine de caractères, elle est réalisée à l'aide de la fonction `strncmp`.

— Si besoin (tableau de chaines de caractères), on boucle.

— Le résultat du traitement est retourné au programme eBPF.

4.3 Cas d'usages

Dans cette section, nous illustrons l'intérêt de SecuHub en présentant plusieurs politiques de mitigation pour des vulnérabiltés récentes. Nous présentons une mitigation pour la CVE-2021-21261 [80] affectant flatpak [37] en 4.3.1, puis la CVE-2019-5736 [79] affectant runc [88] en section 4.3.2, la CVE-2021-3156 [83] affectant sudo en section 4.3.3.1 et enfin la CVE-2021-21315 [81] affectant le package systeminformation [52] de npm en section 4.3.3.

Les attaques présentées dans ce chapitre exploitent des vecteurs d'attaques différents, respectivement une prise de contrôle de flot d'exécution réalisée à travers les variables d'environnements, une élévation de privilèges, un dépassement de tampon (buffer-overflow) et une injection de commandes. Les objets affectés par ces vulnérabilités sont aussi différents, respectivement des variables d'environnement, un démon système lié à un moteur de conteneurisation, un binaire setuid et un service web.

Nous espérons que les exemples d'attaques présentés ici seront complétés par des recherches futures qui exploreront de manière plus systématique les capacités des politiques de mitigation de SecuHub afin d'établir précisément quels types de vulnérabilité peuvent être corrigés avec SecuHub et quels types de vulnérabilités tombent en dehors de son champ d'action.

4.3.1 CVE-2021-21261 affectant flatpak

Flatpak [37] est un logiciel permettant de créer des sandboxes logicielles. Il est notamment utilisé pour mettre en place des environnements virtuels dans le cadre du déploiement logiciel. Ce logiciel est affecté par la vulnérabilité CVE-2021-21261 [80]. Cette vulnérabilité possède un indice CVSS v3 (Common Vulnerability Scoring System version 3) de 8.8 ce qui indique que la vulnérabilité est sérieuse.

Dans le cadre des sandboxes flatpak, il est possible pour un processus sandboxé de créer une nouvelle sandbox avec les mêmes paramètres de sécurité (ou des paramètres plus restreints) que sa sandbox de base à l'aide de la commande `flatpak-spawn`. Il est notamment possible de mettre en place des variables d'environnement pour la nouvelle sandbox à l'aide de l'option `--env=ENV_VAR=VALUE`.

Techniquement, la nouvelle sandbox est créée à partir d'un processus de l'hôte plus tard mis dans les bonnes namespaces avec des droits restreints afin de le transformer en processus de base de la sandbox. Le point d'entrée de la sandbox est alors exécuté. Toutefois, ces variables d'environnement, qui peuvent arbitrairement être mises en place par l'utilisateur pas de confiance sont appliquées au processus appartenant initialement à la namespace de l'hôte. Cela peut mener à des attaques.

Ainsi, quand des variables d'environnement comme `LD_PRELOAD` (qui permet d'exécuter un code additionnel avant le code normal du processus) sont mises en place à l'aide de `--env=LD_PRELOAD=/path/to/hack.so`, le binaire arbitraire `hack.so` s'exécute *dans l'hôte* puisque ce processus appartient à la namespace de l'hôte. On est donc en mesure depuis une sandbox d'exécuter du code arbitraire dans l'hôte. Ceci permet de passer outre les restrictions mises en place par la sandbox, ce qui constitue une élévation de privilèges.

Il est possible de mitiger cette attaque au niveau noyau à l'aide d'une politique de mitigation SecuHub attachée au hook `BPRM_CHECK_SECURITY`, qui est exécuté au lancement d'un programme. Pour mitiger cette vulnérabilité, nous rendons impossible de lancer le binaire `flatpak-spawn` avec un argument sous la forme `--env=ENV_VAR=BAD_ENV` où `BAD_ENV` est une variable d'environnement sensible, par exemple `LD_LIBRARY_PATH`, `LD_PRELOAD`, `LD_AUDIT`... comme montré dans le code 4.3. Cette politique de mitigation est simple et ne requiert qu'un seul helper dynamique SecuHub : `STRING_HELPER` que nous avons présenté en section 4.2.6.3. Les fonctions support utilisées sont `STRING_GET_TOKEN` qui permet de récupérer un token depuis une chaine de caractère selon des délimiteurs et `STRING_BINNAME` qui récupère le chemin du binaire actuellement exécuté. Alternativement, il aurait été possible de mitiger cette attaque en filtrant `envp` (et pas `argv`).

```
void* argsfpk[] = {(void*)0, "/usr/bin/flatpak-spawn"};
if(secuhub_helper(STRING_HELPER, HASH_STRING, STRING_BINNAME, argsfpk) != 1)
    return RET_ALLOW; // If the prgm is not flatpak-spawn -> OK
char badEnv[] = "LD_LIBRARY_PATH;LD_PRELOAD;LD_AUDIT;(...)";
void* argsmatch[] = {ALL_ENV, badEnv, "--env=", "="};
// Is any env (delimited by "-env" and "=") in the forbidden list?
return secuhub_helper(STRING_HELPER, HASH_STRING, STRING_GET_TOKEN,
   argsmatch) >= 0)? RET_DENIED: RET_ALLOW;
```

Code 4.3 – Politique de mitigation pour la CVE-2021-21261

4.3.2 CVE-2019-5736 affectant runc

Nous avons déjà présenté en section 3.4.2 la CVE-2019-5736 affectant runc, comment cette vulnérabilité pouvait être exploitée et comment l'approche SNAPPY pouvait corriger cette vulnérabilité. Toutefois, l'approche montrée n'utilisait pas de helper générique : tout le traitement était réalisé dans un helper dynamique spécifique à cette vulnérabilité. Nous montrons dans le code 4.4 en quoi l'utilisation de helpers dynamiques *génériques* permet une mitigation plus claire et simple à l'aide de ces "briques réutilisables" que sont les helpers dynamiques.

Nous avons déjà indiqué que pour bloquer l'attaque, il suffisait de bloquer le vecteur d'attaque, ce qui pouvait être fait sans bloquer de comportement légitime. Il suffit ici d'exécuter une politique sur le hook `FILE_OPEN` qui regarde si le logiciel runc (ici installé au chemin `/usr/local/sbin/runc`) est ouvert en écriture depuis l'intérieur du conteneur. Dans le code 4.4, le helper `FILE_MODE` qui appartient à `FILE_HELPER`, vérifie si le binaire exécuté est ouvert en écriture et le second, `FILE_PATH`, qui appartient également à `FILE_HELPER`, vérifie si le binaire exécuté est bien runc. Si c'est le cas, on a effectivement une tentative d'attaque : un conteneur ne devrait *jamais* tenter de réécrire sur runc. En cas de détection d'une telle attaque, il est possible de déclencher une alerte avec `ALERT_ONCE`, qui appartient à `STRING_HELPER` et de bloquer cette attaque avec `return RET_DENIED`. Cette politique bloque donc bien la CVE en ne contient que 5 lignes de code, faciles à comprendre et à auditer, en comparaison avec les 195 lignes de code que comprennent le correctif officiel. Cette faille n'augmente pas non plus la taille des conteneurs contrairement au correctif officiel [105].

```
if(secuhub_helper(FILE_HELPER, HASH_FILE, FILE_MODE, {}) & FMODE_WRITE) &&
   secuhub_helper(FILE_HELPER, HASH_FILE, FILE_PATH, {(void*)0,
   "/usr/local/sbin/runc"}) == 1    {
```

```
    secuhub_helper(STRING_HELPER, HASH_STRING, ALERT_ONCE, {PR_ERR,
    "Attempt to exploit CVE-2019-5736 blocked."}))
    return RET_DENIED;
    }
return RET_ALLOWED;
```

Code 4.4 – Politique de mitigation pour la CVE-2019-5736

4.3.3 Attaques en espace utilisateur CVE-2021-3156 et CVE-2021-3177

Puisque les politiques de mitigation SNAPPY sont exécutées au niveau noyau en réaction à des événements LSM, il n'est en théorie pas possible de détecter des attaques qui seraient uniquement exécutées en espace utilisateur. Toutefois dans beaucoup de cas, nous sommes tout de même en mesure d'anticiper les tentatives d'attaques en nous aidant de l'état du programme, des entrées utilisateur, des packets réseaux reçus. Il est possible dans certains cas d'utiliser certains de ces indicateurs pour détecter que le programme est sur le point de tenter une attaque spécifique. Dans de tels cas, il nous est possible d'écrire des politiques de mitigation pour bloquer cette attaque *avant* que le vecteur d'attaque ne soit exploité. Dans cette section, nous montrons à la fois un cas favorable et un cas défavorable de CVE exploitée en espace utilisateur pour montrer clairement ce qui peut être réalisé à l'aide des politiques de mitigation de SecuHub et ce qui tombe en dehors de son champ d'action.

4.3.3.1 CVE-2021-3156 affectant sudo

La faille CVE-2021-3156 [83] affectant la commande système sudo est une vulnérabilité sévère (score CVSS v3 de 7.8). Cette vulnérabilité est d'autant plus critique que le logiciel sudo est utilisé dans un très grand nombre d'environnements. Les détails techniques de l'exploitation de cette vulnérabilité sont présentés dans [96]. Pour synthétiser cette attaque, lorsqu'une commande est exécutée via sudo, les arguments de cette commande sont initialement concaténés et les métacaractères sont échappés à l'aide d'antislashs (`"\"` devient `"\\"`). Malheureusement cet échappement de caractère n'est pas effectué si sudo est exécuté à partir du lien symbolique `sudoedit -s`. Ainsi, si un argument de la commande sudoedit termine par un simple antislash, quand les arguments sont dé-échappés par sudo, le programme va copier des caractères hors du tampon, résultant en un dépassement de

tampon (heap-based buffer overflow). Ainsi, il est notamment possible d'exécuter du code arbitraire en tant qu'administrateur ou faire crasher arbitrairement le programme. Il est donc nécessaire de mitiger cette vulnérabilité.

Un tel buffer overflow est exploité totalement en espace utilisateur. Il reste toutefois possible de mitiger cette vulnérabilité en bloquant toute tentative d'exécuter sudoedit avec un argument terminant par un antislash.

Ainsi, la figure 4.5 présente la politique de mitigation pour la CVE-2021-3156. Elle est attachée au hook `BPRM_CHECK_SECURITY` et mitige la vulnérabilité en empêchant d'exécuter sudoedit si l'un de ses arguments termine par un antislash. Le premier helper dynamique `SRING_BINNAME` du helper dynamique `STRING_HELPER` vérifie si le binaire exécuté est sudo, le second helper dynamique `STRING_REGEX` de `STRING_HELPER` vérifie si l'un des arguments termine par un antislash, c'est à dire s'il matche une certaine regex. Dans ce cas, la commande n'est pas exécutée.

```
if (secuhub_helper(STRING_HELPER, HASH_STRING, STRING_BINNAME, {(void*)0,
    "/usr/bin/sudo"}) == 1 &&
     secuhub_helper(STRING_HELPER, HASH_STRING, STRING_REGEX,   {(void*)-1,
    "^(.*[^\\])?\\$"}) == 1)
     return RET_DENIED;
return RET_ALLOWED
```

Code 4.5 – Politique de mitigation pour la vulnérabilté CVE-2021-3156

Une fois l'attaque détectée, deux stratégies de mitigation peuvent être utilisées. La plus triviale implique simplement d'empêcher l'exécution de la commande, bloquant ainsi l'attaque, comme montré dans le code 4.5. Une stratégie un peu plus robuste peut être de réécrire les arguments du programme dans la politique de mitigation, c'est à dire remplacer `"\"` par `"\\"` pour empêcher l'attaque et transformer le vecteur d'attaque en un comportement valide. Il faut toutefois noter qu'un tel correctif est légèrement plus complexe à écrire puisque les arguments nouvellement écrits pourraient devenir trop larges pour rentrer entièrement sur la page mémoire, auquel cas il faudrait réécrire les arguments sur plus de pages mémoire.

4.3.3.2 CVE-2021-3177 affectant python

Au contraire de la faille présentée dans le paragraphe précédent, certains exploits purement exploités en espace utilisateur sont effectivement des cas symptomatiques et ne peuvent pas être mitigés avec des politiques de mitigation SecuHub. C'est le cas de la CVE-

2021-3177 [84] affectant Python jusqu'en version 3.9.1 via un dépassement de tampon dans la fonction `c_double.from_param()`, qui traduit la représentation textuelle d'un double en la représentation python d'un double. En effet, cette fonction utilise en interne un buffer statique de 256 octets alors que les `double`s python peuvent potentiellement être plus larges, pouvant créer des buffer overflows. Par exemple, le nombre `1e300` crée un buffer de 301 octets, générant un overflow. Un tel overflow peut notamment générer une exécution arbitraire. Il est donc possible d'utiliser *n'importe* quel code python valide pour générer une attaque. De fait, à moins qu'il soit possible de faire des hypothèses supplémentaires sur le format des inputs auquel cas la mitigation serait spécifique à un exploit précis de cette vulnérabilité, il est impossible d'écrire une politique de mitigation qui bloquerait cette vulnérabilité sans avoir recours à un interpréteur python, ce qui n'est pas une approche réaliste au niveau kernel. Cela rend donc cette vulnérabilité un pire cas pour SecuHub.

4.3.4 CVE-2021-21315 affectant systeminformation (npm)

Dans cette section, nous présentons une politique de mitigation ayant la particularité de devoir être configurée pour l'adapter à l'environnement dans lequel cette vulnérabilité est exploitée. La mitigation nécessite de renseigner les variables de configuration spécifiées dans le fichier JSON reçu du serveur SecuHub à la récupération des politiques (voir la section 4.2.2).

La CVE-2021-21315 [81] affectant le package systeminformation [52] de npm, un package de backend npm permettant de récupérer des informations sur le serveur, permet d'injecter de manière malicieuse du code qui finira par être exécuté par le serveur (score CVSS v2 de 7.8). Plus précisément, si une requête web est en mesure d'une manière ou d'une autre de passer le paramètre `BAD_PARAM` au format `$( cmd )` ou ` cmd ` vers systeminformation, par exemple `mysite.domain/vulnerable_page?BAD_PARAM[]=$(cmd)`, systeminformation va exécuter de manière non souhaitable `cmd` sur le serveur puisque `$( cmd )` ou ` cmd ` permettent d'exécuter des sous-commandes avec Bash.

La politique de mitigation pour la CVE-2021-21315, présentée dans le code 4.6 est attachée au hook LSM `SOCKET_RECVMSG`. Elle vise à rejeter les requêtes à la (aux) page(s) `BAD_PAGE` si le(s) argument(s) `BAD_PARAM` ont une valeur malicieuse, c'est à dire `$( cmd )` ou ` cmd `. Les variables `BAD_PAGE` et `BAD_PARAM` sont des variables qui doivent être configurées localement, comme spécifié dans le fichier JSON. Actuellement, `BAD_PAGE` et `BAD_PARAM` sont simplement récupérées de manière interactive par le client SecuHub mais il serait souhaitable de les récupérer de manière automatique et sure via un composant de

pré-installation de confiance. Ces valeurs sont alors fournies à la politique de mitigation à la compilation grâce à l'option `-D` du compilateur.

```
int the_policy(char* raw_url) {
    char* url = decode_url(raw_url);
    if(!is_web_page(url, BAD_PAGE))
        return 0;
    char* param = strip(get_url_param_array(url, BAD_PARAM));
    // Prevents execution of code through $(...) and `...`
    return regex_match(BAD_PARAM, "$\$\(.*\)|`.*`^") ? RET_DENIED:
   RET_ALLOWED;
}
```

Code 4.6 – Politique de mitigation pour la CVE-2021-21315 (pseudocode)

Cette politique de mitigation utilise les fonctions support `decode_url`, `is_web_page` et `get_url_param` fournies par le helper dynamique `NETWORK_REQUEST_HELPER`, un helper développé pour gérer les requêtes réseau. Les fonctions `strip` et `regex_match` sont fournies par le helper `STRING_HELPER`, déjà présenté dans ce document.

4.4 Comparaison avec l'état de l'art

SecuHub est le premier mécanisme à notre connaissance permettant d'automatiser le déploiement et l'installation de politiques de sécurité programmables de niveau noyau.

Dans cette section, nous présentons les approches les plus similaires à SecuHub

Système de détection d'intrusion Beaucoup de systèmes visent à détecter des intrusions potentielles au niveau système (Host Intrusion Detection System) ou réseau (Network Intrusion Detection System) [66]. Certains de ces systèmes peuvent s'adapter aux conteneurs. Cette détection peut être faite en détectant la signature de logiciels malveillants ou des comportements s'écartant du modèle attendu. Ces menaces détectées génèrent des alertes qui sont remontées au SIEM (Security Information and Event Management). Dans certains cas, les systèmes de détections d'intrusions permettent de répondre aux menaces, par exemple en tuant le processus concerné en plus de déclencher une alerte. Nous avons par exemple déjà présenté Falco [101] en section 2.6.5, qui peut servir d'HIDS. Wazuh [121] est une autre plateforme permettant de réaliser de la détection de menaces et de la réponse sur incident. Wazuh peut notamment être utilisé pour la défense de conteneurs grâce à l'utilisation d'un agent embarqué dans l'hôte ou dans un conteneur

privilégié pour surveiller l'activité des autres conteneurs. Toutefois, par design, les IDS ne détectent les attaques que lorsqu'elles ont déjà eu lieu. Il est alors trop tard d'un point de vue sécurité, le système est déjà compromis et est donc exposé à toutes les conséquences associées. Au contraire, SecuHub bloque la tentative d'exploitation de vulnérabilité *avant* qu'elle ait lieu. Le système n'entre donc jamais dans un état compromis, ce qui est souhaitable d'un point de vue sécurité. Par ailleurs, les systèmes de détections d'intrusion peuvent générer beaucoup de fausses alertes. Dans beaucoup de cas, le taux de fausse alerte est élevé, rendant difficile d'analyser les résultats de ces systèmes. De plus, les IDS ne permettent pas de mettre en place des mitigations définitives pour des vulnérabilités données. Enfin, les IDS ne sont pas forcément placés au bon endroit pour analyser des événements et peuvent donc être incapables de détecter certaines attaques. Le placement proposé par SecuHub dans les hooks LSM fournit un point plus avantageux pour détecter les comportements malveillants.

Antivirus Les antivirus constituent une approche visant à détecter et neutraliser les logiciels malveillants. Les antivirus peuvent soit analyser les fichiers pour trouver des signatures de logiciels malveillants, soit étudier le comportement de code pour tenter d'évaluer si celui-ci est malveillant. Il est alors possible de bloquer l'attaque, par exemple en tuant le processus et empêcher de nouvelles tentatives en supprimant le fichier concerné ou en le mettant en quarantaine. Toutefois les antivirus peuvent générer des fausses alertes en tentant de classifier le comportement de logiciels. Par ailleurs, il existe de nombreuses techniques d'évasion afin d'échapper à la détection de virus [86, 95, 91]. Au contraire, puisque SecuHub permet d'écrire des règles identifiant de manière fiable des tentatives d'exploitations de vulnérabilités précises, il est en mesure de détecter de manière fiable de telles tentatives, sans risque de faux positif. Par ailleurs, les antivirus ont généralement un impact significatif sur les performances du système. SecuHub, au contraire garde un impact sur les performances très limité de par son design. Enfin, l'utilisation d'antivirus reste très marginale en environnement conteneur. Les conteneurs utilisent plus fréquemment des techniques d'analyse d'image [13] et de durcissement de configuration, par exemple via AppArmor ou Seccomp-BPF.

Correction à chaud La correction à chaud (hotpatching) est une approche visant à modifier le code source d'un logiciel alors qu'il est déjà installé, notamment pour corriger des vulnérabilités nouvellement trouvées via le blocage des comportements malicieux. Cer-

taines méthodes réalisent du **hotpatching de niveau applicatif** modifiant directement le logiciel en cause. Cette approche est largement étudiée avec des frameworks comme Katana [98], Hera [77] ou Polus [18], mais restent une protection faible puisqu'il est possible pour des attaquants de désactiver des protections mises en places dans l'espace utilisateur. Le **hotpatching de niveau noyau** permet d'obtenir une meilleure isolation des politiques de sécurité. Cette approche est notamment explorée par les frameworks VULMET [124], Kpatch [93] et Ksplice [60]. Toutefois, le hotpatching de niveau noyau reste une tâche complexe. Un correctif incorrect peut engendrer des crashs du système (kernel panic). De par l'utilisation de helpers dynamiques génériques, SecuHub limite la quantité de code critique à développer et à auditer, limitant ainsi ce type de risque. Par ailleurs, les mécanismes mis en place par ces frameworks sont globaux, ce qui signifie qu'il n'est pas possible de mettre en place des défenses uniquement pour des conteneurs/namespaces particuliers, comme c'est le cas avec SecuHub.

Réduction de surface d'attaque préventive Les approches de réduction de surface d'attaque permettent de limiter les fonctionnalités pouvant être exploitées par un conteneur malveillant et ainsi de limiter les risques d'attaques. Toutefois, par design ces mécanismes ne ciblent pas des attaques particulières mais plutôt des blocs de fonctionnalités. Ces mécanismes sont donc complémentaires avec SecuHub mais ne permettent pas de s'assurer que toutes les vulnérabilités connues d'un conteneur donné sont bien mitigées.

Systèmes d'injection de code et de règles Nous avons présenté plusieurs systèmes d'injection de code de niveau noyau en section 2.6 et d'application de règles de niveau noyau en section 2.7 Ces solutions ne gèrent pas le déploiement et l'installation de telles politiques et ne sont donc pas comparables à SecuHub, mais plutôt à SNAPPY. Cette comparaison a été faite en section 3.5.

4.5 Limitations

Dans ce chapitre, nous avons présenté le design et l'implémentation de SecuHub, un framework permettant de déployer et d'installer de manière automatisée des politiques de mitigations pour CVE. SecuHub atteint effectivement les objectifs fixés en section 4.1.2 mais possède toutefois un certain nombre de limitations à corriger dans le futur :

— Certaines politiques de mitigations doivent être configurées. Actuellement, cette

configuration est faite interactivement. Il serait préférable que les données soient récupérées automatiquement. Cela pourrait être fait à l'aide d'un composant de pré-installation de confiance, chargé de récupérer ces données. Le traitement à réaliser pourrait être spécifié directement dans la spécification JSON de la mitigation.

— Par simplicité de développement, les données configurables sont stockées directement dans le code du binaire de la politique de mitigation eBPF. Dans l'optique de rendre plus robuste ce code, il pourrait être pertinent de stocker ces données dans des maps eBPF, l'approche la plus standard pour ce type d'exercice.

— Il serait possible d'augmenter encore plus le niveau d'abstraction des helpers afin qu'ils puissent être appelés comme des fonctions classiques (par exemple `string_helper.strcmp(STR_ARGV, -1, "test")`). Cela pourrait être réalisé en rajoutant un niveau d'abstraction au helper par exemple à l'aide de X-macros, de manière similaire à la couche d'abstraction entre la convention d'appel de SNAPPY et celle de SecuHub.

— Nous n'avons pas traité la problématique du versionnage des helpers. Celle-ci pourrait être traitée simplement en rajoutant un identifiant de version pour les helpers dynamiques. Il serait alors possible pour l'administrateur de mettre à jour les helpers. De tels helpers devraient être rétro-compatibles pour qu'une telle mise à jour ne perturbe pas le fonctionnement du système. Les politiques eBPF pourraient alors dépendre de helpers à partir d'une version précise. Ceci permettrait d'améliorer en continu les helpers SecuHub et donc leur utilisabilité pratique.

Chapitre 5

Évaluation

Résumé du chapitre *Dans ce chapitre, nous évaluons les impacts en performances des frameworks SNAPPY et SecuHub proposés dans cette thèse. Nous montrons au travers d'un microbenchmark et d'un macrobencmark qu'ils peuvent être utilisés en conditions réelles avec un impact négligeable sur les performances des conteneurs.*

Contents

5.1 Considérations générales

Dans ce chapitre, nous évaluons les performances de SNAPPY et celle des cas d'usage SecuHub présentés. Dans le cadre de ces tests, nous utilisons une machine virtuelle avec 6 cœurs alloués, cadencés à 3.8 GHz. Notre machine virtuelle dispose de 4Go de mémoire vive et utilise un noyau Linux 5.5.0 avec les symboles de débogage noyau activés.

Dans le cadre de l'évaluation de SNAPPY, nous présentons d'abord des microbenchmarks mesurant l'overhead induit par les différents éléments du framework SNAPPY. Nous évaluons alors l'overhead de ce framework dans des cas d'usage plus réalistes.

Dans le cadre de l'évaluation de SecuHub, nous présentons comment les helpers dynamiques génériques influent sur la performance du serveur. Nous montrons alors comment des politiques de mitigations affectent les performances du système.

5.2 Microbenchmark

Dans cette section, nous mesurons comment les différents éléments de SNAPPY impactent les performances du système. Nous mesurons donc l'impact sur les performances induit par l'ajout d'un niveau de namespace, l'ajout d'une politique en fonction de son type et l'ajout d'un appel à un helper dynamique.

5.2.1 Impact en performances des namespaces

La figure 5.1 évalue dans quelle proportion la profondeur de l'arbre de namespaces affecte les performances du système. Dans le cadre de ce benchmark, on applique une politique sur le hook `file_open` à un processus dont on fait varier le niveau de namespacing. On appelle alors l'appel système `open()` qui déclenche ce hook et on mesure le temps d'exécution de cet appel système. Afin d'obtenir des valeurs précises, nous mesurons la moyenne et l'intervalle de confiance à 99% sur dix millions d'exécutions de cette politique.

On constate que la profondeur de l'arbre des namespaces affecte très peu les performances du système. En moyenne, rajouter un niveau de namespace rajoute un overhead de 3ns, négligeable pour la plupart des scénarios. C'est d'autant plus vrai que dans le cas général l'arbre de namespaces garde une profondeur assez faible, généralement en dessous de 4. Ainsi, notre abstraction de namespace n'affecte pas significativement les performances du système.

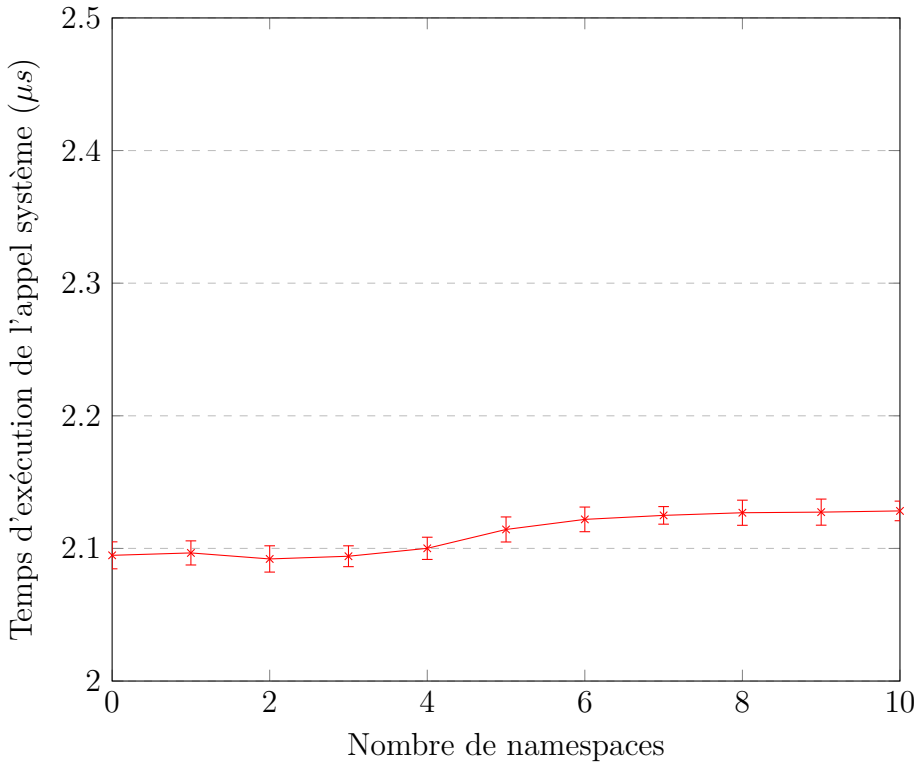


FIGURE 5.1 – Latence de l'appel système `open()` en fonction de la profondeur de l'arbre des namespaces `policy_ns`

5.2.2 Impact en performances des politiques

Un élément important pour évaluer la répartition de l'overhead des frameworks présentés dans ce document est de mesurer dans quelle proportion le rajout de politiques impacte les performances du système. Nous montrons donc dans la figure 5.2 comment le nombre de politiques de sécurité et le nombre de helpers dynamiques impactent les performances du système. Afin de mesurer seulement l'overhead de notre framework et non celui d'une politique particulière, les seuls traitements effectués par les politiques mesurées ici sont l'appel à un certain nombre de helpers dynamiques. Ces helpers dynamiques ne réalisent aucune opération. Comme pour l'expérimentation précédente, nous mesurons le temps d'exécution de l'appel `open()` qui déclenche une fois chacune des politiques. Comme dans l'expérimentation précédente, nous mesurons le temps d'exécution de dix millions de fois la politique pour minimiser les aléas statistiques.

Dans la figure 5.2, les courbes rouge, bleue et verte appellent respectivement 0, 1 et 2

helpers. Dans ce cadre, la latence engendrée par l'ajout d'une politique est de respectivement 49ns (2.56%), 64ns (3.36%) et 77ns (3,89%) pour une politique de sécurité utilisant respectivement 0, 1 et 2 helper(s) dynamiques(s). Ces valeurs correspondent à la différence moyenne entre deux points consécutifs d'une même courbe.

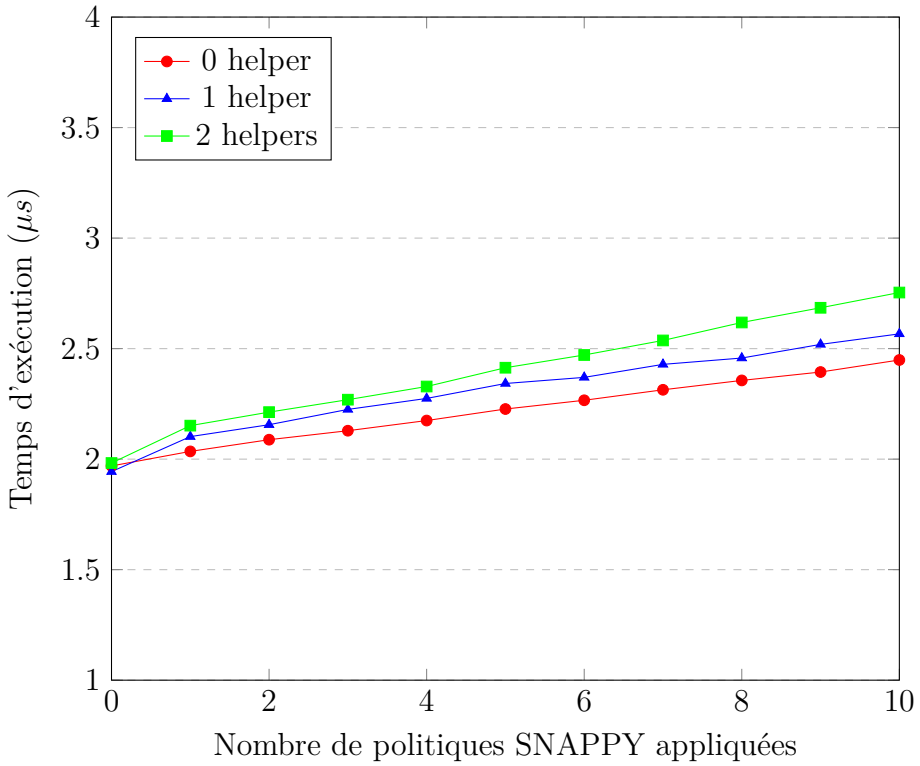


FIGURE 5.2 – Latence de l'appel système `open()` en fonction du nombre et du type des politiques appliquées

Par ailleurs, puisqu'en conditions réelles, seule une petite partie du temps processeur est utilisé pour des appels système qui peuvent déclencher l'exécution de politiques SNAPPY, c'est à dire qu'une partie du temps va être dépensé en espace utilisateur et une autre dans des appels systèmes qui ne généreront pas de politiques SNAPPY, l'impact en performance réel est bien plus faible que les valeurs montrées dans ce microbenchmark, qui peut donc être considéré comme une limite maximale sur les impacts en performances. Ceci sera montré plus en détails dans la suite de notre évaluation. Finalement, puisque les namespaces permettent d'appliquer les politiques à seulement un sous-ensemble du système, l'overhead est totalement nul dans le reste du système. L'overhead moyen est donc encore plus faible que les valeurs annoncées.

5.2.3 Impact en performances des helpers dynamiques

Nous étudions en table 5.1 comment l'appel à des helpers dynamiques SNAPPY impacte les performances des politiques. La figure montre le temps moyen d'exécution de différents helpers eBPF. On constate que dans tous les cas, l'appel se fait en moins de 400ns, ce qui est extrêmement rapide.

Afin d'obtenir ces résultats, nous attachons des politiques contenant un seul appel à un helper dynamique, au hook `file_open`. Nous réalisons alors un appel à `open()`déclenche ces politiques. Nous comparons le temps d'exécution de politiques contenant uniquement l'appel à divers helpers dynamiques à un scénario où une politique vide attachée à ce même hook et ne faisant appel à aucun helper dynamique est mise en place. Pour limiter les aléas statistiques, ces mesures sont moyennées sur dix millions d'appels de politiques. Le temps obtenu dans la table 5.1 est le temps moyen d'exécution de différents helpers dynamiques obtenu par soustraction des temps d'exécution des politiques appelant et de ceux n'appelant aucun helper dynamique.

Fonction support	`STRING_BINNAME`	`STRING_GET_TOKEN`	`STRING_GLOB`	`STRING_STRCMP`
Exec. time (ns)	107	132	343	172

TABLE 5.1 – Temps d'exécution moyen des helpers de `STRING_HELPER`

Puisque les traitements complexes sont réalisés par les helpers dynamiques, la grande majorité du temps d'exécution des politiques est passé dans les helpers dynamiques et il est donc possible d'approximer le temps d'exécution de politiques complexes par le temps passé dans les helpers. Par exemple, la politique mitigeant la CVE-2021-21261 fait appel aux helpers `STRING_BINNAME` et `STRING_GET_TOKEN`, son temps d'exécution peut être estimé via la table 5.1 à $107 + 132 = 239ns$. Cette valeur est très proche de la valeur réelle mesurée (231ns).

La mise en place d'une politique pour la CVE-2021-21261, attachée au hook LSM `BPRM_CHECK_SECURITY` induit donc un overhead estimé à 239ns à chaque fois qu'un programme est lancé dans la (les) namespace(s) surveillé(es), étant donné que le lancement d'un programme déclenche ce hook donc cette politique. Ainsi, même si on lançait 1000 programmes par seconde dans cette namespace, ce qui est une valeur très élevée l'overhead sera seulement de $1000 \times 239ns/s \simeq 0.02\%$ pour ce conteneur, ce qui reste totalement négligeable. Même en prenant le cas d'une politique plus complexe attachée à ce même

hook avec un temps d'exécution de 2µs, l'overhead ne sera que de 0.2% dans le cas d'un conteneur démarrant 1000 programmes par seconde.

5.3 Macrobenchmark

Afin d'obtenir une évaluation réaliste des performances de SNAPPY, nous complétons le microbenchmark présenté en section précédente par une évaluation en conditions plus réalistes. Nous mesurons également dans quelle mesure empiler des politiques de sécurité impacte les performances du système.

5.3.1 Impact en performances de politiques sur des scénarios réalistes

La table 5.2 évalue dans quelles proportions les politiques de sécurité présentées dans le chapitre 3 affectent les performances du système. Nous évaluons ces impacts sur les performances pour deux scénarios différents : celui d'une compilation d'un noyau linux avec la configuration `defconfig` et celui du listing récursif d'un gros dossier à l'aide de `ls -aRl`. Les deux benchmarks ont été effectués dans un conteneur Docker pour lequel `stdout` et `stderr` ont été désactivés pour éviter de mesurer des opérations non liées. Pour réduire les biais statistiques, nous prenons la moyenne sur 100 évaluations dans le cadre de la compilation Linux et 1000 dans le cas du `ls -aRl`. Nous mesurons également l'intervalle de confiance à 99%.

Nous mesurons le temps d'exécution des deux scénarios dans deux cas de figure différents : dans le premier cas, la politique appliquée est celle mitigeant la CVE-2019-5736, présentée en 3.4.2 et dans le second cas c'est la politique garantissant l'unicité des connections réseaux, présentée en section 3.4.1 qui est appliquée. Nous pouvons comparer ces valeurs à un scénario où le framework SNAPPY est totalement désactivé et un autre scénario où SNAPPY est initialisé mais où aucune politique de sécurité n'est appliquée.

Dans les deux scénarios, le binaire exécuté ouvre un grand nombre de fichiers mais ne réalise aucun accès réseau. Par conséquent, la politique garantissant l'unicité de la connexion réseau n'ajoute aucune latence en comparaison du scénario où aucune politique SNAPPY n'est mise en place, au bruit statistique près.

	Compilation noyau linux	`ls -aRl`
SNAPPY désactivé	195.33 ± 0.12	3.296 ± 0.006
Aucune politique SNAPPY	195.57 ± 0.11	3.300 ± 0.003
Mitigation pour CVE-2019-5736	195.76 ± 0.12	3.306 ± 0.003
Unicité de connexion réseau	195.54 ± 0.12	3.301 ± 0.004

TABLE 5.2 – Temps moyen d'exécution des commandes en fonction des politiques SNAPPY appliquées. Résultat : temps moyen (s) ± intervalle de confiance à 99%

Au contraire, la politique liée à la mitigation de la CVE-2019-5736 est proche du scénario pire cas puisqu'elle est déclenchée à chaque ouverture de fichier et une compilation de noyau tout comme la commande `ls -aRl` ouvrent un grand nombre de fichiers. Malgré tout, l'impact en performances reste ici très faible avec un overhead de seulement (0.09%± 0.09%) dans le cadre de la compilation du noyau Linux et 0.03%(±0.02%) dans le cadre du `ls -alr`.

Ainsi, on constate que l'impact en performances de SNAPPY est très négligeable en conditions réelles et est donc adapté au déploiement en production.

De manière similaire à la figure précédente, nous évaluons dans la table 5.3 l'impact sur les performances des politiques de mitigation SecuHub dans le cadre de différents scénarios. Afin d'obtenir un overhead plus facilement mesurable qu'en figure 5.2, les politiques sont appliquées 1000 fois dans cette figure, c'est-à-dire qu'à chaque déclenchement du hook LSM monitoré dans les bonnes namespaces, la politique est exécutée 1000 fois. Pour le reste, nous utilisons la même méthodologie que pour la table précédente. Les politiques évaluées dans cette table sont : pas de politique de mitigation, application d'une politique de mitigation pour la CVE-2021-21261 et application d'une politique de mitigation pour la CVE-2019-5736. Nous réutilisons les mêmes scénarios qu'en figure 5.2 pour l'évaluation.

	Pas de pol.	1000× CVE-2021-21261	1000× CVE-2019-5736
Compil. Linux	147.99 ± 0.21	151.19 ± 0.29	210.42 ± 0.35
`ls -alR`	4.813 ± 0.031	4.800 ± 0.037	6.320 ± 0.034

TABLE 5.3 – Temps d'exécution de commandes (s ± IC à 99%) pour 1000 politiques de mitigation

Les résultats montrent que si l'on rapporte le temps d'exécution des commandes à

l'application d'une seule politique, c'est à dire comme en conditions réelles, l'overhead est très faible, par exemple la politique de mitigation pour la CVE-2019-5736 induit un overhead en performances de seulement 0.03% pour le scénario `ls -alR`. Ces résultats sont donc cohérents avec la table précédente.

En conclusion, pour un scénario avec un nombre réaliste de politiques de mitigations, SecuHub peut être utilisé de manière quasi-transparente.

5.3.2 Impact de l'empilage de politiques

Pour terminer cette évaluation, nous montrons en figure 5.3 comment le nombre de politiques de mitigation affecte les performances du système en conditions réelles en mesurant comment le nombre de requêtes servies par seconde par un serveur web apache2 évolue en fonction du nombre de politiques de mitigation appliquées. Ce benchmark utilise l'utilitaire `ab` avec 100 requêtes simultanées. Nous prenons le temps moyen et l'intervalle de confiance à 99% pour l'exécution de 100 rafales (bursts) de 100 000 requêtes. La politique utilisée ici est celle mitigeant la CVE-2019-5736 affectant runc, décrite en section 4.3.2. Chaque politique est chargée sur le hook `FILE_OPEN`, et est donc exécutée dès qu'un fichier est ouvert. Quand aucune politique n'est appliquée, le serveur peut traiter près de 8000 requêtes par seconde. On constate que même en appliquant 100 fois cette politique, il n'est pas possible de mesurer un overhead significatif dans notre intervalle de confiance à 99%. Il faut une valeur extrême de 1000 politiques pour mesurer un overhead significatif de 16.4%, c'est à dire 0.016% par politique.

Puisque dans des scénarios classiques, le nombre de politiques reste généralement beaucoup plus faible de l'ordre de l'unité ou la dizaine de politiques par conteneur et qu'une partie importante des politiques est appliquée sur des hooks exécutés bien moins fréquemment que `FILE_OPEN`, l'impact en performances de SNAPPY et de SecuHub est très faible en conditions réelles.

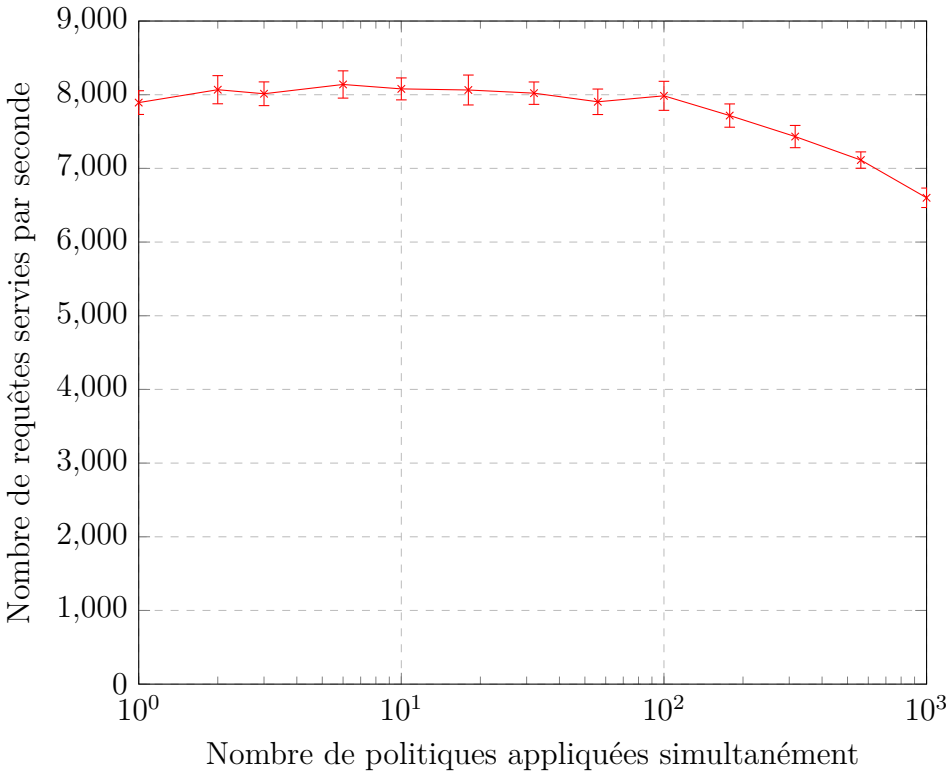


FIGURE 5.3 – Nombre de requêtes servies par le serveur en fonction du nombre de politiques appliquées simultanément

5.4 Conclusion

Nous avons montré dans cette section que les frameworks SNAPPY et SecuHub pouvaient être utilisés pour faire à la fois de la mitigation de vulnérabilité et des politiques de réduction de surface d'attaque avec un overhead en performances très faible. Les raisons expliquant les très bonnes propriétés de performances de ces frameworks sont les suivantes :

— Les politiques sont écrites en eBPF et peuvent donc être compilées à la volée (JIT compilation). Ces politiques n'ont donc pas d'overhead en performances par rapport à du code natif. Les helpers dynamiques sont du code natif et peuvent être appelés comme des fonctions classiques.

— L'utilisation d'une architecture basée sur LSM et les namespaces permet d'appliquer des politiques à des points d'exécution très précis permettant de limiter le

contenu des politiques au minimum.

— Les politiques de sécurité sont exécutées sans changement de contexte.

SNAPPY et SecuHub peuvent donc être utilisés y compris dans des environnements où la performance joue un rôle crucial comme les environnements conteneurs.

Chapitre 6

Bilan et futurs travaux

Contents

6.1 Bilan général de cette thèse

Dans cette thèse, nous avons montré de quelle manière l'application de politiques de sécurité programmables de niveau noyau était prometteuse pour améliorer la sécurité des conteneurs. En effet, malgré une recherche très active sur le sujet les approches existantes restent limitées et ne permettent notamment pas à des conteneurs non privilégiés de mettre en place des politiques de sécurité à grain fin. Par ailleurs, l'application de politiques de sécurité programmables reste minoritaire en production en comparaison des systèmes de règles et des configurations. Ainsi, le but de cette thèse a été de permettre à l'utilisateur d'appliquer simplement des politiques de sécurité programmables de niveau noyau avec un impact sur les performances très faibles, permettant notamment de réaliser de la mitigation de vulnérabilités. De telles mécanismes de sécurité peuvent améliorer significativement la sécurité des conteneurs.

Afin d'atteindre cet objectif, nous avons étudié les différentes formes de virtualisation pour comprendre leurs spécificités. Nous nous sommes particulièrement intéressés à la conteneurisation, puisque cette forme de virtualisation partage le noyau avec l'hôte. De fait le noyau est en mesure de mettre en place de la sécurité pour les conteneurs. Puisque le noyau a accès à toutes les informations sémantiques du conteneur, ces politiques peuvent être très finement grainées.

Nous avons alors présenté un tour d'horizon de la sécurité système et conteneurs. Après une introduction sur les failles de sécurité, les techniques de défenses contre les attaques et une application aux failles conteneurs, nous avons présenté une nouvelle taxonomie des défenses pour les conteneurs. Nous avons classifié les différents frameworks de l'état de l'art dans cette nouvelle taxonomie en faisant apparaitre leur forces et limitations et nous avons montré en quoi les approches basées sur le code étaient une piste de recherche prometteuse et pourquoi la programmabilité du noyau pouvait être utilisée pour améliorer la sécurité du système.

Nous avons présenté le design et l'implémentation de SNAPPY, un nouveau framework permettant au processus et aux conteneurs, y compris non privilégiés, de mettre en place des politiques de sécurité à grain fin de niveau noyau. Ces politiques sont empilables et peuvent être chargées à chaud pour protéger des namespaces/conteneurs, permettant ainsi de minimiser leur surface d'attaque et d'améliorer leur niveau de sécurité. Nous avons intégré SNAPPY avec le runtime OCI afin de pouvoir appliquer SNAPPY à l'intégralité du cycle de vie du conteneur.

Nous avons démontré que SNAPPY pouvait effectivement être utilisé en pratique pour améliorer la sécurité des conteneurs en étudiant le cas d'usage de la mitigation à chaud de vulnérabilités CVEs pour conteneurs. Après avoir montré pourquoi ce cas d'usage était particulièrement pertinent, nous avons présenté le design et l'implémentation du framework SecuHub, qui apporte une réponse à cette problématique en permettant la distribution et l'installation de politiques de mitigations pour des CVEs existantes, applicables à des conteneurs individuels. Nous avons également illustré les capacités de mitigation de SecuHub en présentant des politiques de mitigations pour des CVE existantes. De telles politiques de mitigations n'affectent pas les fonctionnalités légitimes des conteneurs ainsi protégés.

Nous avons par ailleurs montré que grâce à son design, l'utilisation de SNAPPY et de SecuHub n'engendrait qu'un surcoût en performances très faible, ce qui rend ces frameworks particulièrement adaptés pour la protection des conteneurs.

Les travaux menés durant cette thèse ouvrent des perspectives intéressantes, présentées dans la section suivante. Nous pensons que répondre à ces problématiques pourrait constituer un pas important vers une meilleure sécurisation des conteneurs et des environnements système.

6.2 Pistes de recherche

6.2.1 Futures lignes de recherche SNAPPY

Dans l'optique d'améliorer l'utilisation de SNAPPY, plusieurs pistes de recherche peuvent être explorées. Certaines d'entre elles ont déjà été partiellement explorées dans ce mémoire de thèse, d'autres restent à investiguer.

Intégration au noyau Linux Si le code de SNAPPY est aujourd'hui mis en open-source [10], il n'a pas encore été proposé à la communauté Linux. L'amélioration du code de SNAPPY en vue de son intégration dans SNAPPY permettrait d'utiliser ce framework pour des cas d'usages réels.

Évaluation plus poussée Afin de mesurer et de démontrer l'intérêt pratique du déploiement de SNAPPY, une future piste de recherche consiste à développer des politiques pour protéger des conteneurs parmi ceux les plus utilisés dans l'industrie. Cela permettrait de mesurer en pratique quelles mitigations de vulnérabilités et quels types de réduction

de privilèges peuvent être mises en place à l'aide de SNAPPY et ce qui tombe hors de son domaine d'application.

Adaptation des modules LSM Une autre piste de recherche restant à explorer est la transformation de modules LSM existants en helpers SNAPPY. L'application de règles LSM se ferait alors en chargeant des politiques eBPF appelant les modules LSM devenus des helpers SNAPPY et des couches d'abstraction pourraient éventuellement être fournies pour que l'utilisation puisse être faite de manière inchangée par rapport à l'utilisation actuelle des modules LSM existants. Une telle injection de modules LSM dans SNAPPY permettrait de déporter la gestion du namespacing à SNAPPY, plus simple à implémenter que directement dans le module et d'empiler des modules LSM de manière transparente si besoin. Nos résultats préliminaires semblent montrer qu'une telle approche est réalisable en pratique avec seulement des modifications mineures sur le code du LSM, notamment l'exportation d'interface, l'évitement de symboles non-exportées et le stockage des données indépendamment du `lsmblob`.

6.2.2 Futures lignes de recherche SecuHub

L'approche SecuHub ouvre la voie à de nombreuses pistes de réflexion, qui restent ouvertes à ce jour. Nous présentons les principales dans cette section :

Analyse exhaustive Nous ne prétendons pas avoir étudié de manière exhaustive la taxonomie complète des vulnérabilités qui peuvent être corrigées à l'aide de politiques de mitigation SecuHub et lesquelles tombent hors de son champ d'action. Il serait donc nécessaire d'explorer plus systématiquement les capacités de mitigation de SecuHub pour établir ses limites. Cela permettrait notamment d'évaluer le pourcentage de vulnérabilités qui peuvent être corrigées en pratique selon la catégorie de vulnérabilité. Une telle mesure permettrait de motiver l'intérêt de SecuHub en pratique et encouragerait son adoption.

Helper Backtrace Nous investiguons actuellement l'implémentation d'un helper récupérant la backtrace du processus qui déclenche l'opération LSM protégée par une politique de mitigation SecuHub. Un tel helper permet d'exécuter des politiques de sécurité seulement pour des adresses précises. Ainsi, il est possible de n'exécuter le code qu'à des points précis du logiciel, afin d'améliorer encore la finesse des politiques. Cela permet aussi de ne pas exécuter de politiques aux autres adresses, permettant de gagner en performances.

Intégration au noyau Linux Nous rappelons que l'intégration de SNAPPY dans le noyau Linux permettrait d'utiliser SecuHub pour protéger des environnements sans avoir besoin d'utiliser un noyau personnalisé contenant SNAPPY. Cela permettrait d'utiliser SecuHub dans des environnements standards. Nous pensons que le cas d'usage SecuHub justifie la nécessité d'intégrer une namespace orientée sécurité similaire à `policy_ns` ainsi que des framework similaires à SNAPPY. Il serait donc intéressant d'améliorer la base de code de SNAPPY afin de rendre possible une telle intégration dans le noyau.

Utilisation de SecuHub avec un client SecuHub qui n'est pas de confiance Puisque nous avons présenté l'utilisation de SecuHub en environnement conteneurs, nous avons pu faire l'hypothèse que le client SecuHub était de confiance. Afin de permettre l'intégration de SecuHub à d'autres environnements, il serait nécessaire de relâcher cette hypothèse pour certains autres environnements. Seuls le serveur SecuHub et le noyau seraient alors considérés comme de confiance et les autres composants seraient considérés comme non trustés. Les approches techniques qui permettraient d'authentifier les politiques de sécurité malgré l'étape de compilation sont les suivantes :

— **Cross-Compilation des politiques et des helpers depuis le serveur SecuHub** Il est possible de cross-compiler les politiques et les helpers pour le client SecuHub depuis le serveur SecuHub. Dans ce cadre,le serveur SecuHub signe le code binaire des politiques. Le client SecuHub peut alors les charger sans avoir besoin de les compiler. Si un client malveillant tente de modifier des données, les signatures vérifiées dans le noyau seront invalides et le noyau refusera donc de charger ces politiques.

— **Chaines de compilation reproductibles** Une autre approche pour maintenir la compilation au niveau du client est d'utiliser des chaines de compilation reproductibles [55]. Ainsi, il est possible pour le serveur SecuHub de prévoir le hash obtenu après compilation et de le partager au noyau de manière sécurisée. Le noyau est donc en mesure de s'assurer que les politiques ont été correctement compilées avant de les charger.

— **Compilation depuis le noyau** Il est enfin possible de déléguer la compilation des politiques au noyau, simplifiant le design du client. Cela implique soit de réimplémenter une chaine de compilation depuis le noyau ou d'invoquer depuis le noyau une chaine de compilation en espace utilisateur à l'aide de l'API usermode-helper [59].

Avec ces mécanismes, il est possible de s'assurer que les politiques proviennent bien du serveur SecuHub et n'ont pas été modifiées.

Afin de s'assurer que le client SecuHub qui ne serait pas de confiance charge bien la politique demandée, c'est à dire pour éviter le scénario où un client SecuHub compromis prétend installer une politique `A` et installe en fait une politique `B` provenant également du serveur SecuHub, mais sans rapport avec `A` donc pouvant donner un faux sentiment de sécurité, nous proposons les approches suivantes :

— **Vérification interface noyau** L'utilisateur est en mesure de vérifier les politiques de sécurité chargées dans sa namespace et les helpers dynamiques chargés dans le noyau à l'aide d'interfaces noyaux dédiées. Ces interfaces noyau ne passent pas par le client SecuHub et sont donc de confiance. Il est donc possible de vérifier que le client SecuHub a bien chargé les bonnes politiques. Cette vérification peut être faite manuellement ou de manière automatisée.

— **Partage de secret** L'utilisateur pourrait partager avec le noyau un secret représentant le nom de la politique à charger avant de faire appel au client SecuHub pour installer cette politique. Si un client SecuHub malveillant tentait de charger une autre politique, le noyau pourrait le détecter et refuser le chargement, avertissant incidemment l'utilisateur de ce problème. Ce scénario mérite d'être approfondi et n'a pas été étudié en détails.

Cas d'usages additionnels Finalement, bien que les cas d'usages des conteneurs présentés ici soient prometteurs, nous pensons qu'il serait également possible d'appliquer l'approche à d'autres environnements avec des bénéfices potentiels similaires. C'est notamment le cas des téléphones sous Android. Les constructeurs, ne maintiennent typiquement pas plus de quelques années les systèmes d'exploitation de leurs téléphones. C'est d'autant plus problématique que les utilisateurs ne sont pas en mesure de mettre à jour leur version d'Android par eux même à moins de rooter leur téléphone, ce qui nécessite des connaissances techniques et n'est donc pas à la portée de la plupart des utilisateurs. Ainsi, deux mois après la sortie d'Andoid 9, 44,7% des téléphones tournaient encore sous Android 6 ou moins. Puisque Android avait cessé de maintenir ces versions anciennes d'Android, les vulnérabilités trouvées à ce stade n'étaient plus corrigées et le téléphone pouvait donc devenir rapidement très vulnérable. Nous pensons donc que l'approche SecuHub par la mitigation de CVE pourrait permettre d'améliorer la sécurité de ces environnements en ouvrant d'autres moyens prometteurs de traiter les vulnérabilités dans les environnements

Linux, dont Android.

Publications de l'auteur et contribution à l'opensource

Conférences internationales avec actes et comité de lecture

- SecuHub : Distributing Kernel-level Security Policies for Container Vulnerabilities Mitigation. **Maxime Bélair**, Sylvie Laniepce, and Jean-Marc Menaud. En cours de soumission.
- SNAPPY : programmable kernel-level policies for containers. **Maxime Bélair**, Sylvie Laniepce, and Jean-Marc Menaud. In Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC '21). Mars 2021, Gwangju / Virtual, South Korea. DOI : `https://doi.org/10.1145/3412841.3442037` [12]
- Leveraging Kernel Security Mechanisms to Improve Container Security : a Survey. **Maxime Bélair**, Sylvie Laniepce, and Jean-Marc Menaud. In Proceedings of the 14th International Conference on Availability, Reliability and Security (ARES '19). Août 2019. Canterbury, Royaume-Uni DOI : `https://doi.org/10.1145/3339252.3340502` [11]

Conférence nationale avec actes et comité de lecture

- Container interaction with host OS for enhanced security : a survey. **Maxime Bélair**, Sylvie Laniepce, Jean-Marc Menaud. Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS 2019), Juin 2019, Anglet, France

Brevets

- Procédé de sécurisation d'un appel système, procédé de mise en place d'une politique de sécurité associée et dispositifs mettant en œuvre ces procédés. **Maxime Bélair**, Sylvie Laniepce. N° de dépôt : 20 2095. Déposé le 20/05/2020 (numéro référence extension PCT FR2021050860).
- Procédé et module d'installation d'un programme de mitigation dans le noyau d'un équipement informatique. **Maxime Bélair**, Sylvie Laniepce, N° dépôt : 20 2314. Déposé le 2/03/2021.

Code OpenSource

— Code source de SNAPPY : `https://github.com/Orange-OpenSource/SNAPPY`.

BIBLIOGRAPHIE

[1] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker : Lightweight virtualization for serverless applications. In *17th {usenix} symposium on networked systems design and implementation ({nsdi} 20)* (2020), pp. 419–434.

[2] Aiken, M., Fähndrich, M., Hawblitzel, C., Hunt, G., and Larus, J. Deconstructing process isolation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness* (New York, NY, USA, 2006), MSPC '06, Association for Computing Machinery, p. 1–10.

[3] AMD. Secure virtual machine architecture reference manual. *AMD Publication 33047* (2005).

[4] Anand, P. A presentation of ebpf. `https://opensource.com/article/17/9/intro-ebpf`, 2017.

[5] Anchore. Grype's repository. `https://github.com/anchore/grype`, 2021.

[6] Aquasecurity. Trivy's repository. `https://github.com/aquasecurity/trivy`, 2021.

[7] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., and Fetzer, C. Scone : Secure linux containers with intel sgx. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (2016), OSDI'16, pp. 689–703.

[8] Avino, G., Malinverno, M., Malandrino, F., Casetti, C., and Chiasserini, C. F. Characterizing docker overhead in mobile edge computing scenarios. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems* (New York, NY, USA, 2017), HotConNet '17, Association for Computing Machinery, p. 30–35.

[9] Bauer, M. Paranoid penguin : An introduction to novell apparmor. *Linux J.* (2006).

[10] Bélair, M. Snappy github repository. https ://github.com/Orange-OpenSource/SNAPPY, 2021.

[11] Bélair, M., Laniepce, S., and Menaud, J.-M. Leveraging kernel security mechanisms to improve container security : A survey. In *Proceedings of the 14th International Conference on Availability, Reliability and Security* (2019), ARES '19.

[12] Bélair, M., Laniepce, S., and Menaud, J.-M. Snappy : Programmable kernel-level policies for containers. In *SAC 2021 : 36th ACM/SIGAPP Symposium On Applied Computing* (Gwangju / Virtual, South Korea, 2021).

[13] Brady, K., Moon, S., Nguyen, T., and Coffman, J. Docker container security in cloud computing. In *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)* (2020), IEEE, pp. 0975–0980.

[14] Bugnion, E., Nieh, J., and Tsafrir, D. Hardware and software support for virtualization. *Synthesis Lectures on Computer Architecture 12*, 1 (2017), 1–206.

[15] Burns, B., Beda, J., and Hightower, K. *Kubernetes : up and running : dive into the future of infrastructure*. O'Reilly Media, 2019.

[16] Cabodi, G., Giorgio, D. A., and Minì, G. Os-level virtualization with linux containers : process isolation mechanisms and performance analysis of last generation container runtimes.

[17] Calavera, D., and Fontana, L. *Linux Observability with BPF : Advanced Programming for Performance Analysis and Networking*. O'Reilly Media, 2019.

[18] Chen, H., Yu, J., Chen, R., Zang, B., and Yew, P.-C. Polus : A powerful live updating system. In *29th International Conference on Software Engineering (ICSE'07)* (2007), pp. 271–281.

[19] Chen, Q., and Bridges, R. A. Automated behavioral analysis of malware : A case study of wannacry ransomware. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)* (2017), IEEE, pp. 454–460.

[20] Cho, H., Park, J., Kang, J., Bao, T., Wang, R., Shoshitaishvili, Y., Doupé, A., and Ahn, G.-J. Exploiting uses of uninitialized stack variables in linux kernels to leak kernel pointers. In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)* (2020).

[21] Cilium. Cilium repository. https ://github.com/cilium/cilium, 2021.

[22] Compastié, M., Badonnel, R., Festor, O., and He, R. From virtualization security issues to cloud protection opportunities : An in-depth analysis of system virtualization models. *Computers & Security 97* (2020), 101905.

[23] Containers, L. Lxc repository. `https://github.com/lxc/lxc`, 2021.

[24] Cook, K. Kernel address space layout randomization. *Linux Security Summit* (2013).

[25] Corbet, J. The apparmor debate begins. `https://lwn.net/Articles/181508/`, 2006.

[26] Corbet, J. Reconsidering unprivileged bpf. Reconsidering unprivileged BPF, 2019.

[27] Corbet, J. Landlock (finally) sets sail. https ://lwn.net/Articles/859908/, 2021.

[28] Dangaard Brouer, J. ebpf maps. `https://prototype-kernel.readthedocs.io/en/latest/bpf/ebpf_maps.html`, 2021.

[29] De Benedictis, M., and Lioy, A. Integrity verification of docker containers for a lightweight cloud environment. *Future Generation Computer Systems 97* (2019), 236–246.

[30] de Raadt, T. Pledge(). In *Hackfest 2015* (2016).

[31] developpers, L. ebpf helpers – linux manual page.

[32] Edge, J. Making containers safer. https ://lwn.net/Articles/796700/, 2019.

[33] Falliere, N., Murchu, L. O., and Chien, E. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response 5*, 6 (2011), 29.

[34] Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)* (2015), IEEE, pp. 171–172.

[35] Findlay, W., Barrera, D., and Somayaji, A. Bpfcontain : Fixing the soft underbelly of container security. *arXiv preprint arXiv :2102.06972* (2021).

[36] Findlay, W., Somayaji, A., and Barrera, D. Bpfbox : Simple precise process confinement with ebpf. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop* (2020), CCSW'20, p. 91–103.

[37] Flatpak. Flatpak repository. `https://github.com/flatpak/flatpak`, 2021.

[38] Foundation, F. S. Apparmor labeling system. https://elixir.bootlin.com/linux/v5.14/source/security/apparmor/label.c, 2021.

[39] Foundation, F. S. cgroups man page. https://man7.org/linux/man-pages/man7/cgroups.7.html, 2021.

[40] Foundation, F. S. Chroot man page (2). http://man7.org/linux/man-pages/man2/chroot.2.html, 2021.

[41] Foundation, F. S. Clone3() syscall man page. https://man.archlinux.org/man/clone3.2.en#clone3(), 2021.

[42] Foundation, F. S. Namespaces man page (7). https://man7.org/linux/man-pages/man7/namespaces.7.html, 2021.

[43] Foundation, F. S. Yama kernel documentation. https://www.kernel.org/doc/html/v4.15/admin-guide/LSM/Yama.html, 2021.

[44] Foundation, O. Owasp website. https://owasp.org/, 2021.

[45] Foundation, T. L. Linux sched.h. https://elixir.bootlin.com/linux/v5.14/source/include/uapi/linux/sched.h#L25, 2021.

[46] freedesktop.org. Presentation of seccomp bpf. https://dri.freedesktop.org/docs/drm/userspace-api/seccomp_filter.html, 2017.

[47] Google. Gvisor github repository. https://github.com/google/gvisor, 2021.

[48] Gough, J. J., and Gough, K. J. *Compiling for the. Net Common Language Runtime*. Prentice Hall PTR, 2001.

[49] Graf, T. ebpf - rethinking the linux kernel. In *QCon London 2020* (2020).

[50] Hardy, N. The confused deputy : (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review 22*, 4 (1988), 36–38.

[51] Hebbal, Y., Laniepce, S., and Menaud, J.-M. Virtual machine introspection : Techniques and applications. In *2015 10th International Conference on Availability, Reliability and Security* (2015), pp. 676–685.

[52] Hildebrandt, S. Systeminformation package. https://www.npmjs.com/package/systeminformation, 2021.

[53] Hofkvmlbreich, A. Docker components explained. https ://alexander.holbreich.org/docker-components-explained/, 2018.

[54] Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., and Miller, D. The express data path : Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies* (2018), pp. 54–66.

[55] https ://reproducible builds.org/. Reproducible builds. https ://reproducible-builds.org/, 2021.

[56] inc., D. Dockerhub website. `https://hub.docker.com/`, 2021.

[57] iovisor. Bcc repository. https ://github.com/iovisor/bcc, 2021.

[58] Iwaniuk, A., and Popławski, B. Cve-2019-5736 : Escape from docker and kubernetes containers to root on host. https ://blog.dragonsector.pl/2019/02/cve-2019-5736-escape-from-docker-and.html, 2019.

[59] Jones, M. Invoking user-space applications from the kernel. implementation and use of the usermode-helper api. https ://developer.ibm.com/articles/l-user-space-apps/, 2010.

[60] Kaashoek, M., and Arnold, J. Ksplice : Automatic rebootless kernel updates. *Frans Kaashoek* (2009).

[61] kernel development community, T. Lsm bpf programs. `https://www.kernel.org/doc/html/latest/bpf/bpf_lsm.html`, 2021.

[62] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. kvm : the linux virtual machine monitor. In *Proceedings of the Linux symposium* (2007), vol. 1, Dttawa, Dntorio, Canada, pp. 225–230.

[63] Kotakanbe. Vuls' repository. `https://github.com/future-architect/vuls`, 2021.

[64] Kumar, R., and Thangaraju, B. Performance analysis between runc and kata container runtime. In *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)* (2020), IEEE, pp. 1–4.

[65] Larsen, B., Debes, H. B., and Giannetsos, T. Cloudvaults : Integrating trust extensions into system integrity verification for cloud-based environments. In *European Symposium on Research in Computer Security* (2020), Springer, pp. 197–220.

[66] Liao, H.-J., Lin, C.-H. R., Lin, Y.-C., and Tung, K.-Y. Intrusion detection system : A comprehensive review. *Journal of Network and Computer Applications 36*, 1 (2013), 16–24.

[67] Liu, L. [cve-2019-5736] : Runc uses more memory during start up after the fix, 2019.

[68] Liu, P., Ji, S., Fu, L., Lu, K., Zhang, X., Lee, W.-H., Lu, T., Chen, W., and Beyah, R. Understanding the security risks of docker hub. In *European Symposium on Research in Computer Security* (2020), Springer, pp. 257–276.

[69] Lu, H., Matz, M., Hubicka, J., Jaeger, A., and Mitchell, M. System v application binary interface. *AMD64 Architecture Processor Supplement* (2018).

[70] Lu, H., Matz, M., Hubicka, J., Jaeger, A., and Mitchell, M. System v application binary interface. *AMD64 Architecture Processor Supplement* (2018).

[71] Manco, F., Lupu, C., Schmidt, F., Mendes, J., Kuenzer, S., Sati, S., Yasukata, K., Raiciu, C., and Huici, F. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 218–233.

[72] McCanne, S., and Jacobson, V. The bsd packet filter : A new architecture for user-level packet capture. In *USENIX winter* (1993), vol. 46.

[73] Merkel, D. Docker : Lightweight linux containers for consistent development and deployment. *Linux J. 2014*, 239 (Mar. 2014).

[74] Microsoft. Integrity policy enforcement lsm (ipe). https ://lwn.net/Articles/816952/, 2020.

[75] Morris, J., Smalley, S., and Kroah-Hartman, G. Linux security modules : General security support for the linux kernel. In *USENIX Security Symposium* (2002), ACM Berkeley, CA, pp. 17–31.

[76] Nam, J., Lee, S., Seo, H., Porras, P., Yegneswaran, V., and Shin, S. Bastion : A security enforcement network stack for container networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (2020), pp. 81–95.

[77] Niesler, C., Surminski, S., and Davi, L. Hera : Hotpatching of embedded real-time applications. In *28th Network and Distributed System Security Symposium (NDSS)* (2021).

[78] NIST. Cve-2017-9150 detail. `https://nvd.nist.gov/vuln/detail/CVE-2017-9150`, 2017.

[79] NIST. Cve-2019-5736 detail. https://nvd.nist.gov/vuln/detail/CVE-2019-5736, 2019.

[80] Nist. Cve-2021-21261 detail. https://nvd.nist.gov/vuln/detail/CVE-2021-21261, 2021.

[81] Nist. Cve-2021-21315 detail. https://nvd.nist.gov/vuln/detail/CVE-2021-21315, 2021.

[82] NIST. Cve-2021-30465 detail. https://nvd.nist.gov/vuln/detail/CVE-2021-30465, 2021.

[83] Nist. Cve-2021-3156 detail. https://nvd.nist.gov/vuln/detail/CVE-2021-3156, 2021.

[84] NIST. Cve-2021-3177 detail. https://nvd.nist.gov/vuln/detail/CVE-2021-3177, 2021.

[85] NIST. Cve-2021-3490 detail. https://nvd.nist.gov/vuln/detail/CVE-2021-3490, 2021.

[86] Oberheide, J., Bailey, M., and Jahanian, F. Polypack : an automated online packing service for optimal antivirus evasion. In *Proceedings of the 3rd USENIX conference on Offensive technologies* (2009), pp. 9–9.

[87] OpenContainers. Open container initiative runtime specification repository. https://github.com/opencontainers/runtime-spec/blob/master/spec.md, 2021.

[88] opencontainers. Runc repository. https://github.com/opencontainers/runc, 2021.

[89] Orman, H. The morris worm : A fifteen-year perspective. *IEEE Security & Privacy 1*, 5 (2003), 35–43.

[90] Pahl, C. Containerization and the paas cloud. *IEEE Cloud Computing 2*, 3 (2015), 24–31.

[91] Panagopoulos, I. Antivirus evasion methods. Master's thesis, University of Piraeus, 2020.

[92] Perez-Botero, D., Szefer, J., and Lee, R. B. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 International Workshop on Security in Cloud Computing* (New York, NY, USA, 2013), Cloud Computing '13, Association for Computing Machinery, p. 3–10.

[93] Poimboeuf, J. Kpatch github's repository, 2021.

[94] Potdar, A. M., Narayan, D., Kengond, S., and Mulla, M. M. Performance evaluation of docker container and virtual machine. *Procedia Computer Science 171* (2020), 1419–1428.

[95] Poulios, G., Ntantogian, C., and Xenakis, C. Ropinjector : Using return oriented programming for polymorphism and antivirus evasion. *Blackhat USA* (2015).

[96] Qualys. Baron samedit : Heap-based buffer overflow in sudo (cve-2021-3156). https://lwn.net/ml/oss-security/20210126181453.GA4184@localhost.localdomain/, 2021.

[97] Quay. Clair's repository. https://github.com/quay/clair, 2021.

[98] Ramaswamy, A., Bratus, S., Smith, S. W., and Locasto, M. E. Katana : A hot patching framework for elf executables. In *2010 International Conference on Availability, Reliability and Security* (2010), pp. 507–512.

[99] Randazzo, A., and Tinnirello, I. Kata containers : An emerging architecture for enabling mec services in fast and secure way. In *2019 Sixth International Conference on Internet of Things : Systems, Management and Security (IOTSMS)* (2019), IEEE, pp. 209–214.

[100] Sailer, R., Zhang, X., Jaeger, T., et al. Design and implementation of a tcg-based integrity measurement architecture. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13* (2004), SSYM'04, pp. 16–16.

[101] Salamero, J. Kubernetes runtime security with falco and sysdig. https://www.cncf.io/wp-content/uploads/2019/12/Kubernetes-Runtime-Security-with-Falco-and-Sysdig.pdf, 2019.

[102] Salaün, M. [rfc patch v14 00/10] landlock lsm. https ://lore.kernel.org/lkml/20200224160215.4136-1-mic@digikod.net/, 2020.

[103] Salaün, M. Landlock lsm : toward unprivileged sandboxing. *Linux Security Summit* (2017).

[104] SANS. How do we define responsible disclosure ?

[105] Sarai, A. Cve-2019-5736 patch commit. https://github.com/opencontainers/runc/commit/0a8e4117e7f715d5fbeef398405813ce8e88558b, 2019.

[106] Schaufler., C. Lsm stacking - what you can do now and what's next.

[107] Shahzad, A., and Litchfield, A. Virtualization technology : Cross-vm cache side channel attacks make it vulnerable.

[108] Shen, Z., Sun, Z., Sela, G.-E., et al. X-containers : Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2019), ASPLOS '19, ACM, pp. 121–135.

[109] Shu, R., Gu, X., and Enck, W. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (CODASPY'17)* (2017), pp. 269–280.

[110] Singh, K. Kernel runtime security instrumentation. In *Linux Security Summit North America 2019* (2019).

[111] Smalley, S., Vance, C., and Salamon, W. Implementing SELinux as a Linux security module. *NAI Labs Report 1* (2001), 43.

[112] Soppelsa, F., and Kaewkasi, C. *Native Docker Clustering with Swarm*. Packt Publishing Ltd, 2016.

[113] Souppaya, M., Morello, J., and Scarfone, K. Application container security guide. Tech. rep., National Institute of Standards and Technology, 2017.

[114] Sultan, S., Ahmad, I., and Dimitriou, T. Container security : Issues, challenges, and the road ahead. *IEEE Access 7* (2019), 52976–52996.

[115] Sun, Y., Safford, D., Zohar, M., Pendarakis, D., Gu, Z., and Jaeger, T. Security namespace : Making linux security frameworks available to containers. In *27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, 2018), USENIX Association, pp. 1423–1439.

[116] Tanenbaum, A. S., and Bos, H. *Modern operating systems*. Pearson, 2015.

[117] Team, L. D. Memory management. `https://elixir.bootlin.com/linux/v5.14/source/Documentation/x86/x86_64/mm.rst`, 2021.

[118] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C., Anderson, A. V., Bennett, S. M., Kagi, A., Leung, F. H., and Smith, L. Intel virtualization technology. *Computer 38*, 5 (2005), 48–56.

[119] Van Moolenbroek, D. C., Appuswamy, R., and Tanenbaum, A. S. Towards a flexible, lightweight virtualization alternative. In *Proceedings of International Conference on Systems and Storage* (2014), pp. 1–7.

[120] Venners, B. *Inside the Java Virtual Machine*. McGraw-Hill, Inc., USA, 1996.

[121] Wazuh. Wazuh repository. `https://github.com/wazuh/wazuh`, 2021.

[122] Wist, K., Helsem, M., and Gligoroski, D. Vulnerability analysis of 2500 docker hub images. In *Advances in Security, Networks, and Internet of Things*. Springer, 2021, pp. 307–327.

[123] world stats, I. Internet world stats. https ://www.internetworldstats.com/stats.htm, 2021.

[124] Xu, Z., Zhang, Y., Zheng, L., Xia, L., Bao, C., Wang, Z., and Liu, Y. Automatic hot patch generation for android kernels. In *USENIX Security Symposium* (2020).

[125] Yi, B., Wang, X., Li, K., Huang, M., et al. A comprehensive survey of network function virtualization. *Computer Networks 133* (2018), 212–262.

[126] Young, E. G., Zhu, P., Caraza-Harter, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. The true cost of containing : A gvisor case study. In *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing (HotCloud'19)* (USA, 2019), HotCloud'19, USENIX Association, p. 16.

TABLE DES FIGURES

Liste des codes

Liste des tableaux

DOCTORAT BRETAGNE LOIRE / MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Titre : Défense contre les attaques à l'aune des nouvelles formes de virtualisation des infrastructures

Mot clés : Sécurité, Conteneurs, SNAPPY, SecuHub, Noyau, Programmable

Résumé : La conteneurisation est une forme de virtualisation de niveau système d'exploitation présentant de bonnes propriétés de performances et de simplicité de déploiement ; elle facilite la réutilisation du code. Les conteneurs sont donc massivement utilisés aujourd'hui. Toutefois, de par leur grande surface d'attaque et les vulnérabilités qu'ils peuvent souvent contenir, les conteneurs posent de nouveaux enjeux de sécurité. Les nombreuses approches défensives existantes ne suffisent pas à répondre à toutes leurs problématiques de sécurité.

Dans cette thèse, nous montrons que la programmabilité du noyau permet de déployer des services de sécurité innovants pour améliorer la sécurité des conteneurs. Après avoir montré les spécificités des environnements conteneurs et leur problématiques et opportunités de sécurité, nous présentons le design et l'implémentation de SNAPPY, une nouvelle infrastructure logicielle permettant de mettre en place des politiques de sécurité programmables à grain fin de niveau noyau, particulièrement adaptée à la protection des conteneurs. Nous présentons également SecuHub, une nouvelle infrastructure logicielle de distribution unifiée de politiques de mitigation pour CVE (Common Vulnerabilities and Exposures), permettant donc aux conteneurs de se protéger simplement contre les vulnérabilités connues. Nous montrons finalement que le surcoût en performance de SecuHub et SNAPPY est minimal.

Title: Defense against attacks with regard to new virtualisation technics

Keywords: Security, Containers, SNAPPY, Secuhub, Kernel, Programmable

Abstract: Containerization is an OS-level virtualization technique providing good performances, ease of deployment and code reusability properties. Containers are therefore massively used nowadays. However, due to their big attack surface and to the vulnerability they may contain, containers bring new security challenges. The numerous existing defensive approaches are not sufficient to respond to all their security issues.

In this thesis, we show that kernel programmability allows to deploy innovative security services to improve the security of containers. After showing the specificities of containers environments and associated security challenges and opportunities, we present the design and implementation of SNAPPY, a new framework allowing to setup fine-grained programmable kernel security policies notably suitable to protect containers. We also present SecuHub, a new framework enabling to distribute CVE mitigation policies, allowing containers to protect themselves against known vulnerabilities. We finally show that SNAPPY and SecuHub can be used with a very low performance overhead.