Conception et implémentation d'un langage dédié à l'introspection de machine virtuelle

Lionel Hemmerlé[1], Guillaume Hiet[1], Frédéric Tronel[1], Pierre Wilke[1], Jean-Christophe Prévotet[2]
[1]CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA
[2]INSA Rennes : UnivRennes, INSA Rennes, CNRS, IETR –UMR 6164, F-35000 Rennes

***Résumé*—Afin de pouvoir protéger des machines virtuelles (VM) contre des intrusions, il est possible de placer un système de détection d'intrusion directement au niveau de l'hyperviseur. Dans cette situation, même si un attaquant arrive à compromettre intégralement une VM, il ne pourra pas désactiver l'IDS. Une difficulté se posant alors est celle du fossé sémantique : l'IDS à accès a toutes les informations propres à une VM, mais ne sait pas comment les interpréter. Pour franchir ce fossé sémantique, nous proposons alors de créer un langage permettant à la VM de communiquer à l'hyperviseur comment interpréter les données et dans quelles situations il est nécessaire de lever une alerte. Nous avons alors implémenté deux attaques et leur détection au niveau de l'hyperviseur grâce à l'envoi d'informations de la VM sur les régions de sa mémoire qui doivent êtres surveillées par l'hyperviseur.**

I. Introduction

Afin de pouvoir réagir à une attaque, il est nécessaire de pouvoir la détecter. Cela nécessite de recourir à un système de détection d'intrusion (IDS) que l'on peut placer sur les systèmes que l'on souhaite protéger. Si un attaquant arrive à accéder a un haut niveau de privilège sur une machine infectée, il peut alors essayer de désactiver un IDS ou de lui communiquer des informations fausses, le rendant alors incapable de détecter l'intrusion. Ainsi, pour garantir le fonctionnement d'un IDS, il est nécessaire de le protéger en l'isolant du système qu'il surveille. Pour cela, nous pouvons utiliser les extensions de virtualisation du processeur.

Une extension de virtualisation est une extension matérielle présente sur un processeur qui permet de simuler le fonctionnement d'un ou plusieurs processeurs virtuels et d'exécuter sur ces processeurs virtuels un système d'exploitation. On appelle un tel système simulé une Machine Virtuelle (VM). Le fonctionnement des différentes VM ainsi que leurs interactions avec le matériel sont alors contrôlées par un composant logiciel que l'on appelle un hyperviseur. Nous proposons ainsi de placer un IDS au niveau d'un hyperviseur pour le protéger. Dans cette situation, même si une VM est compromise par un attaquant, l'hyperviseur ainsi que l'IDS peuvent continuer de s'exécuter normalement.

Une fois placé dans un hyperviseur, l'IDS a accès à tout le contenu de la mémoire de la VM contrôlée par l'hyperviseur et à ses échanges avec les composants matériels, mais l'IDS perd les abstractions fournies par le système d'exploitation de la VM et doit donc être capable de reconstruire l'état interne de cette VM (par exemple retrouver la liste des processus exécutés) ainsi que les événements qui y ont lieu (par exemple la création d'un nouveau processus). Il s'agit du problème du fossé sémantique[1].

II. État de l'art

Pour franchir ce fossé sémantique, différentes approches ont été proposées. Il est possible de modifier le code du noyau de la VM pour y intégrer des *hooks* [2] qui permettent de détecter certains événements ayant lieu dans la VM et de communiquer des informations à l'hyperviseur pour que celui-ci puisse déclencher des alertes. Cette méthode nécessitant de modifier le code de la VM, un attaquant peut alors détecter la présence de ces *hooks* et peut essayer de les contourner. Westphal et al. [3] ont développé une autre approche : leur outil permet à un utilisateur d'indiquer à l'hyperviseur dans quelles situations lever des alertes grâce à des programmes écrits dans un langage dédié. Leur approche nécessite cependant d'utiliser des fichiers de configurations supplémentaires pour chaque système d'exploitation utilisé dans les VMs afin de pouvoir effectivement franchir le fossé sémantique. Pour se passer de tels fichiers, il est possible d'utiliser des *hyperupcalls* [4] : une VM envoie des programmes à l'hyperviseur, qui peut alors les exécuter pour obtenir des informations sur l'état de cette VM. Cette approche est très intéressante, mais a été utilisée principalement pour optimiser l'utilisation des ressources matérielles par l'hyperviseur (par exemple en déterminant les zones de la mémoire inutilisées de la VM) et n'a donc pas été développée dans le but de détecter des intrusions. Ainsi, il est possible pour un attaquant de rendre les programmes exécutés par l'hyperviseur inopérants.

III. Approche

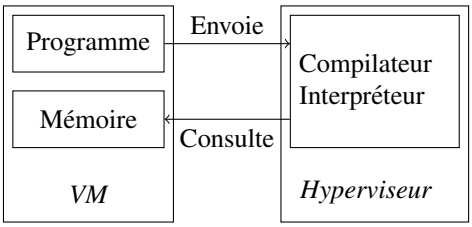

Fig. 1. Fonctionnement de notre système de détection

Nous souhaitons créer un langage permettant à un hyperviseur de détecter des intrusions dans une VM en exécutant des programmes écrits dans ce langage (Figure 1). Ces programmes doivent alors être envoyés par cette VM à

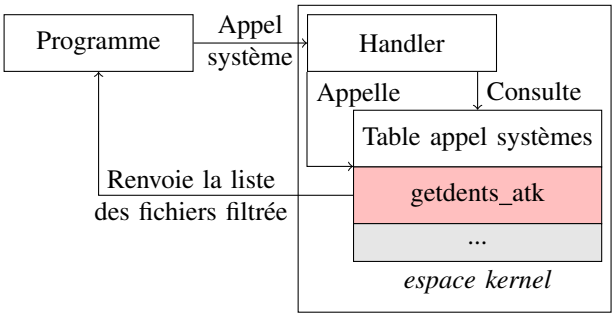

Fig. 2. Manipulation de la table d'appel système

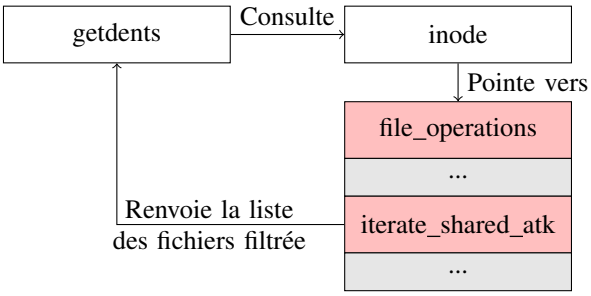

Fig. 3. Manipulation des structures du système de fichier virtuel

l'hyperviseur et doivent contenir toutes les informations (ou les traitements permettant de les obtenir) nécessaires au franchissement du fossé sémantique. Puisque cette approche permet à l'hyperviseur d'exécuter du code fourni directement par des VMs, il faut s'assurer qu'un attaquant ne puisse pas détourner notre langage pour compromettre l'hyperviseur. Pour éviter cela, il faut d'abord restreindre l'envoi de programmes par la VM à une période initiale où elle peut être considérée comme saine. De plus, puisque ces programmes prennent en entrée des données qui peuvent être modifiées par l'attaquant (dans la mémoire de la VM), il faut pouvoir s'assurer que l'hyperviseur ne puisse pas être attaqué en exploitant d'éventuels bogues dans les programmes destinés à franchir le fossé sémantique.

IV. Implémentation

Nous avons utilisé XVisor [5] afin de lancer une VM contenant la version 5.10 du noyau Linux sur un processeur ARM64. Nous avons créés par ailleurs deux rootkits pouvant être exécutés dans la VM afin et permettant de masquer un processus vis à vis de l'espace utilisateur (le processus n'apparaît plus si un utillisateur utilise la commande `ps`) et nous avons modifié XVisor afin que celui-ci puisse détecter les deux rootkits.

A. Rootkits

Lorsqu'un utilisateur utilise la commande `ps` afin de pouvoir lister les processus existants, cette dernière lit le contenu du dossier `/proc`, qui contient un dossier pour chaque processus en cours d'exécution. Pour pouvoir masquer un processus, il faut alors faire en sorte que ces dossiers ne soient pas visibles par un utilisateur. Pour cela, plusieurs approches sont possibles. Les approches utilisées dans les rootkits que nous avons implémentés sont les suivantes :

- Modifier la table d'appel système,
- Modifier les structures utilisées par le Système de Fichier Virtuel (VFS).

1) Modification de la table d'appel système: Lorsqu'un programme exécuté en espace utilisateur souhaite lister le contenu d'un répertoire, il doit effectuer l'appel système `getdents64`. Lorsqu'un appel système est effectué, le noyau consulte une structure nommée table d'appel système contenant un pointeur vers la fonction à exécuter pour chaque appel système.

Ainsi, pour masquer un processus à l'espace utilisateur, il est possible de remplacer le pointeur de la fonction correspondant à l'appel système `getdents64` par un pointeur vers une autre fonction contrôlée par l'attaquant (Figure 2). Cette fonction va alors pouvoir modifier le résultat de l'appel système original si l'utilisateur souhaite accéder au contenu du dossier `/proc` afin d'y supprimer les entrées correspondant aux processus qu'un attaquant souhaite masquer.

Pour pouvoir détecter cette attaque au niveau de l'hyperviseur, il faut que ce dernier puisse détecter les modifications de la table d'appel système, il faut donc qu'il soit capable de détecter lorsque la VM essaie d'écrire dans certaines zones de sa mémoire.

2) Modification du VFS: Le VFS est un composant du noyau Linux permettant de fournir une interface uniformisée pour tous les systèmes de fichiers implémentés. Ce dernier fonctionne en utilisant notamment des *inodes* qui permettent de représenter un fichier ou un dossier et les structures *file_operations* qui contiennent des pointeurs vers les fonctions permettant d'exécuter différentes opérations sur un inode (par exemple ouvrir un fichier ou lire le contenu d'un répertoire).

Ainsi, lorsqu'un utilisateur effectue l'appel système `getdents64` décrit précédemment, le noyau accède à l'*inodes* correspondant au fichier `/proc` pour accéder à la structure `file_operations` correspondante et ensuite exécuter la fonction pointée par le champ `file_operations.iterate_shared` qui permet alors de lister le contenu de ce répertoire.

Le rootkit que nous avons implémenté remplace alors ce champ `file_operations.iterate_shared` par un pointeur vers une fonction qui masque les entrées relatives aux processus que l'on souhaite masquer (Figure 3).

Pour pouvoir détecter cette attaque au niveau de l'hyperviseur, il faut que ce dernier puisse détecter les modifications de la structure `file_operations` utilisées par l'inode représentant le répertoire `/proc`.

B. Détection

Afin de détecter les deux attaques que nous avons implémentées, nous avons modifié XVisor afin qu'il puisse détecter les écritures dans la mémoire de la VM.

Pour cela, nous avons utilisé les structures utilisées par le mécanisme de traduction d'adresse.

Lorsqu'un processus intéragit avec la mémoire (par exemple en manipulant des pointeurs), il ne manipule pas des adresses physiques, mais des adresses virtuelles qui sont ensuite traduites par le processeur grâce aux registres `TTBR0_EL1` et `TTBR1_EL1` et à des tables dédiées à cette traduction d'adresse qui sont créées par le noyau Linux pour chaque processus. Dans la configuration utilisée par le noyau Linux, `TTBR0_EL1` pointe vers la table de niveau 0 accessibles pour traduire les adresses utilisées par le processus en cours d'exécution, et `TTBR1_EL1` pointe vers la table de niveau 0 utilisée pour traduire les adresses pointant vers la mémoire du noyau. Les tables de niveau 0 contiennent alors des pointeurs vers des tables de niveau 1, qui pointent vers des tables de niveau 2 qui pointent vers des tables de niveau 3, qui contiennent alors l'adresse physique d'une région contigüe de 4 kilo-octets (appelée page) contenant les données correspond à l'adresse virtuelle initiale. L'adresse physique finale pouvant être obtenue en combinant l'adresse de la page avec les derniers bits de l'adresse virtuelle.

Lorsque le système est exécuté dans une VM, la traduction d'adresse contrôlée par la VM permet d'obtenir une adresse intermédiaire, et une deuxième traduction d'adresse similaire à la précédente est ensuite effectuée par le processeur, mais utilisant cette fois-ci des structures et registres contrôlés par l'hyperviseur afin d'obtenir la véritable adresse physique.

Les entrées des tables utilisées lors de la traduction d'adresse contrôlée par l'hyperviseur contiennent des entrées permettant de déterminer les actions autorisées par la VM avec le contenu de la mémoire (lire, écrire sur la page correspondante, ou exécuter son contenu).

Ainsi, en interdisant les écritures de la VM sur une page donnée, une excception est levée au niveau de l'hyperviseur dès que la VM essaie d'écrire sur cette page. L'hyperviseur peut alors traiter cette exception, et si elle résulte d'une tentative de la VM d'écrire sur la table d'appel système ou la structure `file_operations` utilisées par l'inode du repertoire `/proc`, l'hyperviseur peut lever une alerte pour signaler qu'une attaque à été détectée.

Comme la table d'appel système ou la structure `file_operations` n'ont pas des tailles égales à des multiples de la taille d'une page, des exceptions peuvent ainsi être levées par des écritures légitimes qui ont lieu sur la même page qu'une de ces structures mais pas sur la structure elle-même. Il faut donc que ces écritures puissent s'exécuter normalement même lorsque la protection est activée. Pour cela, lorsque l'hyperviseur détecte une telle situation, il réautorise temporairement les écritures de la VM sur la page concernée par l'exception, puis exécute unqiuement l'instruction de la VM ayant tenté d'écrire sur cette page avant d'interdire à nouveau les écritures de la VM. Si cette instruction à écrit au moins partiellement sur une structure protégée, l'hyperviseur peut alors réécrire les données d'origine avant d'éxécuter à nouveau normalement la VM.

Pour que la protection soit effective, l'hyperviseur a besoin de connaître les adresses de la table d'appel système et de la structure `file_operations` que l'on souhaite protéger. Pour cela, nous avons implémenté au niveau de l'hyperviseur un hypercall permettant à la VM d'envoyer à l'hyperviseur l'adresse virtuelle et la taille d'un structure que l'on souhaite protéger. L'hyperviseur peut alors transformer les adresses virtuelles correspondantes en adresses intermédiaires puis adresses physiques pour déterminer quelles sont les pages devant être surveillées.

Grâce à ces modifications, l'hyperviseur a été capable de lever des alertes pour chacune des deux attaques implémentées, et a également été capable d'annuler les modifications que ces deux attaques ont effectuées sur la mémoire de la VM, permettant ainsi de protéger la VM.

V. Conclusion et travaux futurs

Nous avons implémenté deux rootkits permettant de masquer des processus dans une VM exécutant le noyau Linux, et nous avons modifié l'hyperviseur XVisor afin qu'il puisse détecter les écritures de la VM dans certaines zones de la mémoire afin de pouvoir protéger la VM des deux rootkits implémentés.

Ceci ne permet que de détecter des attaques reposant sur la modification de structures qui sont censées être en lecture seule dans la mémoire du noyau. Elles ne permettent donc pas de détecter des attaques plus complexes, reposant par exemple sur la modification des structures `struct PID` qui sont utilisées pour lister les processus actifs, ou les structures utilisées par l'ordonnanceur du noyau linux (afin de pouvoir exécuter un processus invisible du reste du système).

Pour pouvoir détecter ces attaques, nous souhaitons développer un langage dédié à la detection d'intrusion dans la VM, dont les programmes permettent d'indiquer quels événements doivent être surveillés par l'hyperviseur et de vérifier, lorsque ces événements ont lieu, s'ils correspondent bien à des modifications légitimes (par exemple si le processus en cours d'exécution est bien listé dans toutes les structures de données référençant les processus actifs, ou pour vérifier que l'*effective user id* d'un processus est modifié seulement grâce aux appels systèmes permettant de le modifier pour détecter une élevation de privilège).

Références

[1] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, "Sok: Introspections on trust and the semantic gap," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 605–620.

[2] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, 2008, pp. 233–247.

[3] F. Westphal, S. Axelsson, C. Neuhaus, and A. Polze, "Vmi-pl: A monitoring language for virtual platforms using virtual machine introspection," *Digital Investigation*, vol. 11, pp. S85–S94, 2014, fourteenth Annual DFRWS Conference.

[4] M. Wei and N. Amit, "Leveraging hyperupcalls to bridge the semantic gap: An application perspective." *IEEE Data Eng. Bull.*, vol. 42, no. 1, pp. 22–35, 2019.

[5] *XVisor hypervisor*. [Online]. Available: https://github.com/xvisor/xvisor