Suivi de flux d'information correct sous Linux

Laurent Georget Guillaume Piolle Mathieu Jaume
Frédéric Tronel Valérie Viet Triem Tong Sorbonne Universités,
EPC CIDRE CENTRALESUPELEC/ UPMC, CNRS, LIP6
INRIA/CNRS/Université de Rennes 1 UMR 7606
laurent.georget@irisa.fr

Résumé

À l'échelle des systèmes d'exploitation, le contrôle de flux d'information vise à contrôler la façon dont les données circulent entre des objets tels que les processus, les fichiers, les sockets réseaux, etc. afin de prévenir les fuites et modifications illégales d'information. Nous avons étudié trois implémentations pour Linux : Laminar [3], KBlare [6] et Weir [2] qui ont en commun d'être basées sur l'interface des *Linux Security Modules* (LSM). Cependant, LSM a été conçu à l'origine pour le contrôle d'accès [5] et la question se pose de savoir s'il est utilisable également pour implémenter de manière fiable le contrôle de flux d'information. Nous décrivons ici deux travaux, l'un ayant conduit à vérifier une condition nécessaire, l'autre à proposer une condition suffisante pour répondre à cette question. Nous présentons Rfblare, une version de KBlare permettant un suivi de flux prouvé correct et résistant aux conditions de concurrence entre appels système.

1 Introduction

Dans un système d'exploitation, les processus, les fichiers, les tuyaux, les sockets et les zones de mémoire partagées sont des conteneurs d'information : via leurs fonctions de stockage, traitement ou acheminement, ils participent à la propagation de l'information à l'intérieur du système ainsi que vers et depuis l'extérieur. Cependant, tout flux d'information n'est pas désirable. Par exemple, si un logiciel malveillant exfiltre des données confidentielles hors du système ou bien chiffre un fichier et demande une rançon pour le restituer, le système a échoué à garantir, respectivement, les propriétés de confidentialité et d'intégrité. Le mécanisme traditionnel pour garantir ces propriétés est le contrôle d'accès : en prévenant tout *accès*, même indirect, à des données qu'il serait illégal de retransférer, on prévient naturellement tout flux illégal. Cependant, il semble utile qu'un processus puisse donner à un autre processus l'accès à une donnée confidentielle sans lui permettre de la diffuser à son tour à des conteneurs en lesquels le premier processus n'a pas confiance. Le contrôle de flux d'information se distingue du contrôle d'accès en ce qu'il permet, moyennant une certaine connaissance des flux passés dans le système, de garantir l'absence de flux illégaux tout en étant plus libéral que le contrôle d'accès dans les politiques qu'il permet d'appliquer.

Cette connaissance des flux passés du système est généralement basée sur un mécanisme de propagation de teintes : chaque conteneur est associé à un label initial, une méta-donnée qui indique la classe des informations qu'il contient. Par la suite, à chaque flux d'information, le label de la destination du flux est mis à jour en fonction du label de la source pour matérialiser le fait que la destination peut à présent contenir des données de la source, qui elle-même a pu les hériter d'une autre source. À chaque instant de la vie du système, le label d'un conteneur caractérise l'ensemble des sources qui ont pu contribuer à son contenu. Le contrôle de flux d'information est étudié depuis plusieurs décennies et a donné lieu à de nombreuses implémentations. Nous en avons étudié trois pour Linux : Laminar [3], KBlare [6] et Weir [2]. Laminar et KBlare visent le noyau Linux officiel et Weir les systèmes Android. Ces trois outils implémentent le contrôle de flux d'information en utilisant la propagation de teintes, bien que leurs labels n'aient pas exactement la même sémantique. Un autre point commun est qu'ils sont tous trois implémentés avec le framework *Linux Security Modules* (LSM).

LSM met à disposition des développeurs des champs de sécurité supplémentaires à l'intérieur des structures de données du noyau ainsi que des *crochets* où des modules de sécurité peuvent enregistrer des fonctions pour consulter l'état du noyau et du module (stocké dans les champs de sécurité) et prendre une décision de sécurité avant une opération sensible. Les crochets sont positionnés à l'intérieur du code des appels système, dans le noyau. Nous faisons l'hypothèse que tout flux d'information à l'échelle du système d'exploitation requiert au minimum un appel système. Cette hypothèse est réaliste (et répandue) car seul le noyau peut accéder au matériel directement ainsi qu'à la mémoire de tous les processus. Par conséquent, pour stocker de l'information dans un fichier ou utiliser un canal de communication interprocessus, ou bien pour mettre en place un segment de mémoire partagée, il est nécessaire d'effectuer un appel système. La position des crochets est donc critique pour garantir qu'un mécanisme de contrôle de flux d'information basé sur LSM peut effectivement intercepter tous les flux.

2 Condition nécessaire : présence d'un crochet avant chaque opération causant un flux

De nombreux flux d'information existent dans le noyau Linux et correspondent à des ensembles d'appels système variés. À l'intérieur de ces appels système se trouvent des crochets LSM. Cependant, s'il est possible d'effectuer un appel système générant un flux sans passer par les crochets s'y trouvant, le moniteur de flux d'information n'aura pas l'opportunité d'agir en fonction de ce flux. En effet, le moniteur de flux n'est interrogé qu'au passage dans un crochet. Une condition nécessaire à une propagation correcte des flux d'information est donc qu'il se trouve au moins un crochet le long de tous les chemins d'exécution générant des flux d'information à l'intérieur des appels système. Nous appelons cette propriété la *médiation complète*.

Nous proposons trois contributions en vue de vérifier cette propriété. En premier lieu, nous construisons un modèle du code des appels système sous la forme de graphes de flot de contrôle, produit par le compilateur GCC lui-

même. Nous concevons ensuite une analyse statique reproductible et applicable sur chaque appel système indépendamment. Enfin, nous montrons que cette analyse permet de caractériser quelques problèmes de LSM dans le placement des crochets pour le contrôle de flux d'information.

2.1 Graphes et chemins d'exécution

De manière classique, notre analyse statique considère le code d'un appel système comme un graphe orienté où chaque nœud représente une instruction et où un arc matérialise le fait que l'exécution d'une instruction peut être suivie par l'exécution d'une autre instruction. Les arcs peuvent être étiquetés par une condition de branchement. La représentation des instructions n'est pas celle du C mais d'une représentation intermédiaire du compilateur C de GCC (*Gnu Compilers Collection*) appelé GIMPLE. En effet, La construction de notre modèle et l'exécution de notre analyse statique se fait littéralement à l'intérieur de la chaîne de compilation du noyau car nous avons développé notre analyse comme un plugin du compilateur GCC. Cela nous permet de bénéficier de toutes les représentations internes de GCC et, en un sens, de toute la connaissance que GCC a du code du noyau. Par exemple, le traitement des variables de GCC a déterminé notre modèle de mémoire. Certaines variables sont adressables, et peuvent être modifiées par des pointeurs ou des appels de fonction, tandis que d'autres variables ne voient jamais leur adresse prise. Pour le compilateur, cela signifie qu'il peut les optimiser à sa guise ; pour nous, que ces variables ne peuvent pas changer de valeur hormis par une affectation directe et qu'elles sont donc très utiles à l'analyse statique.

Par rapport au C, le GIMPLE est réduit à une poignée d'instructions basiques et la syntaxe de notre graphe s'en trouve d'autant simplifiée. On peut distinguer six types de nœuds différents : les affectations simples, les affectations à travers un pointeur, les nœuds de jonction, les nœuds phi et les nœuds d'appels de fonction. Les nœuds d'affectation change la valeur d'une variable ou une case mémoire. Les nœuds de jonction sont les seuls nœuds du graphes à avoir plusieurs successeurs, ils représentent les branchements conditionnels. Les nœuds phi suivent les nœuds de jonction possédant plusieurs prédecesseurs et représentent l'affectation des variables dont la valeur dépend du chemin suivi jusqu'alors dans le graphe. Enfin, les nœuds d'appel de fonction représente des instructions qui peuvent modifier de manière arbitraire la mémoire et la variable de retour. Normalement, le graphe comprend aussi des nœuds représentant du code assembleur inclus dans les sources C mais nous modélisons ceux-ci par des nœuds d'appels de fonction.

2.2 Propriété de médiation complète

Il y a exactement un graphe par appel système. Les chemins d'exécution de l'appel système font partie des chemins, au sens de la théorie des graphes, qui commencent à l'entrée de l'appel système (le seul nœud sans prédecesseur) et se terminent au retour de l'appel (un nœud sans successeur). Cependant, tous les chemins du graphe ne sont pas des chemins d'exécution. En effet, certains de ces chemins ne peuvent pas être empruntés lors d'une exécution concrète car ils nécessitent de passer par exemple par deux arcs

étiquetés par des conditions incompatibles entre elles. Comme les crochets LSM sont inclus dans le code du noyau, et non à l'entrée des appels système, il peut exister des chemins dans le graphe qui contournent le crochet LSM mais qui passent dans la branche où le flux est effectué. Si ce chemin est impossible, ce n'est pas un problème, en revanche, s'il est possible, c'est une violation de la propriété de complète médiation. Par conséquent, la précision de notre analyse repose sur la détection correcte de ces chemins impossibles. Enfin, comme dans toute analyse statique, nous devons abstraire l'analyse des boucles car la présence d'une seule boucle dans le code rend le nombre de chemins à analyser infini. L'analyse retourne une surapproximation correcte pour un nombre d'itérations arbitraire de chaque boucle. Afin que tous les crochets LSM et les branches contenant la génération du flux d'information soient bien inclus dans le graphe analysé (et ne soient pas cachés derrière un appel de fonction), notre analyse force l'*inlining* des branches concernées, c'est-à-dire que les appels de fonction intéressantes sont remplacés par le corps de cette fonction dans le graphe représentant l'appel système. Cette opération est faite par le compilateur, donc le code retient sa sémantique.

La propriété de complète médiation peut donc se formuler comme suit :

Propriété 1. *Tout chemin dans le graphe, commençant à l'entrée de l'appel système, se terminant au retour de l'appel, passant par un nœud correspondant à la réalisation du flux d'information mais ne passant par aucun crochet LSM, est un chemin impossible.*

Cette propriété suffit à garantir qu'il n'existe pas de chemin d'exécution esquivant tous les crochets LSM mais effectuant le flux d'information.

2.3 Analyser l'impossibilité d'un chemin

Contrairement aux analyses de flots de données comme *live variables* ou *very busy expressions*, l'analyse statique introduite est dépendante du chemin (*path-sensitive*). Elle est basée sur une notion de configuration qui associe à chaque variable un ensemble de contraintes arithmétiques sur ses valeurs possibles. Pour chaque chemin à analyser, l'analyse commence avec une configuration vide. À chaque nœud, la configuration est modifiée en fonction des propriétés prédictibles sur la sémantique concrète du programme. Certaines instructions comme les affectations ajoutent des contraintes tandis que d'autres comme des appels de fonctions, dont il n'est pas toujours possible de prédire le comportement, relâchent ces contraintes. En particulier, nous considérons qu'un appel de fonction peut modifier arbitrairement la valeur de toute variable adressable. Nous retirons donc de la configuration les contraintes portant sur ces variables lorsqu'un appel de fonction est traversé. Enfin, les étiquettes des arcs empruntés contribuent également à ajouter des contraintes. Un chemin est considéré comme impossible lorsque, pour au moins une variable, l'ensemble de ses contraintes n'est pas satisfaisable. Par exemple, passer par le nœud $x = \&y$ puis $*x = 3$ puis par un arc étiqueté par $y < 2$ met d'abord à jour la configuration avec $\{x = \&y\}$ puis $\{x = \&y, y = 3\}$ puis $\{x = \&y, y = 3, y < 2\}$ qui est clairement une configuration insatisfiable. La satisfiabilité d'une configuration est déterminée par le SMT-solveur Yices [1]. Dès qu'une configuration insatisfiable est obtenue, le chemin analysé est déclaré impossible. Si l'analyse se poursuit

jusqu'à la fin du chemin, alors il est déclaré possible. L'analyse est correcte : les chemins déclarés impossibles ne peuvent pas correspondre à une exécution concrète du programme. Bien sûr, une telle analyse ne peut pas être complète : certains chemins déclarés impossibles sont en réalité possibles. L'analyse que nous proposons est très simple, beaucoup de traits du langage ne sont pas pris en compte et donnent lieu à des surapproximations (relâchement de contraintes) mais elle suffit pour répondre à nos objectifs.

Nous avons appliqué cette analyse à tous les appels systèmes correspondant à des IPC ou à une lecture–écriture dans un fichier ou encore à une création de processus. Nous avons placé dans le code une annotation sur les crochets LSM et les appels de fonction correspondant à la réalisation du flux d'information (par exemple, l'appel de la fonction modifiant le contenu des fichiers, après toutes les vérifications) pour notre plugin. Les résultat que nous avons obtenus identifient un petit ensemble d'exécutions ne satisfaisant pas la propriété de complète médiation. Ces résultats permettent de rajouter ou déplacer des crochets LSM dans le code pour que ces exécutions soient couvertes. Nous pouvons conclure que LSM reste approprié pour le contrôle de flux d'information car les exécutions problématiques ne représentent qu'une faible fraction de toutes celles existantes, et l'analyse permet d'étendre facilement ce framework.

Cette approche démontre également que concevoir des analyses statiques via le compilateur est très bénéfique. Notre analyse porte ainsi sur le code qui sera exécuté, selon la sémantique que lui donne le compilateur, et non directement sur celui qui est écrit. Par exemple, cela permet de bénéficier des représentations internes du compilateur, et en particulier des graphes de flot de contrôle. Nous avons effectué l'analyse sur la version 4.3 du noyau Linux mais elle est reproductible sur toutes. L'implémentation et nos résultats sont disponibles en ligne sur `https://kayrebt.gforge.inria.fr`.

3 Condition suffisante : aucun flux n'échappe au moniteur

Dans la section précédente nous avons montré comment modifier LSM en ajoutant quelques crochets de telle sorte qu'il soit impossible de générer un flux dans un des appels système analysés sans passer par un crochet LSM. Il est donc possible d'implémenter un moniteur de flux basé sur LSM qui surveille tous les flux. Toutefois, la présence d'un crochet sur chaque chemin engendrant un flux n'est pas une condition suffisante pour affirmer qu'aucun flux n'échappe au moniteur.

3.1 Attaques contre les moniteurs de flux

Les moniteurs de flux sont sujets à des *conditions de concurrence* car dans un appel système, la séquence d'opérations ⟨passage dans un crochet LSM, réalisation effective du flux⟩ n'est pas atomique. Imaginons par exemple deux processus, l'un effectuant l'appel système `write` pour écrire dans un fichier et l'autre effectuant l'appel système `read` pour lire depuis ce même

fichier. Une exécution possible est :

	lecteur	**écrivain**
1	passage dans le crochet `read`	—
2	—	passage dans le crochet `write`
3	—	flux du lecteur vers le fichier
4	flux du fichier vers le lecteur	—

Dans cette exécution, il y a un flux indirect de l'écrivain vers le lecteur via le fichier car le flux de l'écrivain vers le fichier a eu lieu avant le flux du fichier vers le lecteur. Cependant, le moniteur de flux d'information n'est appelé que lors du passage dans les crochets LSM et voit par conséquent le flux fichier–lecteur comme antérieur au flux écrivain–fichier.

Nous avons noté ce problème en premier lieu en testant le moniteur de flux d'information Blare [6]. Afin de montrer que ce problème n'est pas limité à Blare mais inhérent à toutes les solutions implémentées avec LSM, nous avons également étudié Laminar [3], un autre moniteur de flux d'information pour le noyau Linux de base, et Weir [2] dédié au noyau Linux Android. Nous avons développé plusieurs attaques contre Blare et porté l'une d'entre elles pour Weir, sous la forme d'applications Android. Cette attaque impliquent deux fichiers, deux processus ainsi qu'un *pipe*. Un processus appelé *émetteur* lit le fichier *source* et écrit son contenu dans le *tube* tandis qu'en parallèle, le processus *récepteur* lit depuis le tube puis écrit les données lues dans le fichier *destination*. On constate que si le récepteur commence à lire le tube avant que l'émetteur n'y écrive, il va se bloquer en attente des données. Cependant, à ce point, il aura déjà traversé le crochet LSM et le flux du fichier source vers lui, et indirectement vers le fichier destination ne sera pas vu. L'usage d'un tube contraint l'ordre des flux car lire depuis un tube vide est bloquant mais il faut noter que la condition de concurrence existerait tout autant en utilisant un fichier standard, elle serait seulement plus difficile à déclencher.

3.2 Contremesures

Un moyen de contrer cette attaque pourrait être de transformer les sections de code s'étendant des crochets LSM aux générations de flux en sections atomiques. C'est l'approche choisie par Laminar pour les appels système `read` et `write`. Cependant, cela peut diminuer grandement les performances de ces appels système en ne permettant pas leur parallélisme, ou pire, peut conduire à des interblocages. Cela n'est pas acceptable pour un système d'exploitation. On pourrait également vouloir rapprocher les crochets LSM des points du code où sont générés les flux d'information, afin d'empêcher les conditions de concurrence. Cela pourrait toutefois être incompatibles avec les autres usages de LSM. Enfin, il n'est pas toujours aisé de dire précisément dans un appel système à quel endroit est effectué le flux d'information. Dans le cas d'un fichier par exemple, s'agit-il du moment où les informations sont copiées du buffer de l'utilisateur au cache en mémoire du fichier ? Ou bien lorsque la taille du fichier est mise à jour ? Ou encore lorsque les informations sont écrites sur le disque ? Pour ces raisons, nous avons décidé d'adopter une approche prudente. Nous considérons que le moniteur de flux d'information peut uniquement avoir connaissance des bornes de la section de code entre

lesquelles le flux peut avoir lieu. La borne ouvrante est le crochet LSM déjà présent (nous avons vérifié dans la section précédente qu'il existe bien un tel crochet dans tous les cas) et ouvre un flux. Le retour de l'appel système peut être pris comme borne fermante et ferme le flux ouvert.

L'algorithme que nous proposons repose sur deux ensembles de flux : les flux déjà réalisés (ayant déjà eu lieu) dans le système et les flux en cours. Au début de la vie du système, ces deux ensembles sont vides. L'ensemble des flux en cours est défini comme la fermeture réflexo-transitive de la relation caractérisant l'ensemble des flux ouverts. C'est l'ensemble des flux s'effectuant en concurrence, comme `read` et `write` de l'exemple. Les flux réalisés dans le système sont mis à jour à chacun des événements d'ouverture et fermeture de flux en calculant la composition de ces flux avec les flux en cours. Par exemple, si l'ensemble des flux réalisés est $\{A \to A, B \to B, A \to B\}$ et l'ensemble des flux ouverts est $\{C \to D, B \to C\}$, alors l'ensemble des flux en cours est $\{C \to D, B \to C, B \to B, C \to C, D \to D, B \to D\}$ et $\{A \to A, B \to B, A \to B, A \to C, A \to D, B \to C, B \to D\}$ correspond alors au nouvel ensemble des flux réalisés. Nous avons démontré que ce résultat correspond à la plus petite surapproximation des flux que l'on puisse calculer qui prenne en compte tous les entrelacements de flux possibles. Autrement dit, chaque flux identifié peut être engendré par une exécution concrète compatible avec la séquence d'événements d'ouverture et de fermeture de flux rencontrés. Les preuves ont été obtenus par l'assistant de preuves Coq [4] à partir d'une description formelle de l'algorithme. Nous avons implémenté cet algorithme dans le moniteur de flux d'information Blare [6] et nous l'avons évalué. Une propriété remarquable de notre moniteur est qu'il peut gérer correctement les flux d'information causés par les mémoires partagées (ou les projection en mémoire de fichiers) qui sont des exemples typiques de flux indirects : si un processus A a une mémoire partagée avec B qui lui-même a une mémoire partagée avec C alors il existe un flux indirect entre A et C qui ne peut pas être détecté en considérant les flux indépendamment les uns des autres. De plus, les mémoires partagées requièrent des appels système uniquement pour mettre en place et détacher la mémoire (les opérations individuelles de lecture–écriture ne nécessitent aucun appel système). Le problème causé par les mémoires partagées est connu et considéré comme difficile, comme l'attestent certains commentaires dans les codes source de Laminar [3] et Blare.

3.3 Implémentation

Nous avons démontré l'utilité et l'implémentabilité de notre algorithme de surveillance des flux dans *Rfblare*, une nouvelle implémentation du suivi de flux pratiqué par Blare. Tout comme Blare, Laminar et Weir, Rfblare est implémenté comme un module LSM. Pour valider notre implémentation, nous avons réutilisé les attaques développées contre Blare. La première attaque est celle mettant en jeu la condition de concurrence sur le tube. Naturellement l'ordre des événements reste identique mais Rfblare y réagit correctement. Lorsque le processus récepteur s'endort en tentant de lire depuis le tuyau vide, le flux reste ouvert. Pendant ce temps, le processus émetteur lit le fichier source et écrit dans le tube. À ce moment, le flux de l'émet-

teur vers le tube est détecté, et comme le flux du tube vers le récepteur est toujours ouvert, Rfblare voit le flux indirect de l'émetteur vers le récepteur. Finalement, à la fin de l'exécution, le flux du fichier source, dont sont originaires les données, vers le fichier destination, où le processus récepteur écrit, est correctement détecté. Nous avons également testé l'attaque exploitant les mémoires partagées en utilisant une situation de test similaire. Enfin, nous avons testé les performances de Rfblare et conclut que ce dernier entraîne un surcoût négligeable dans des cas d'usage classiques comme la compilation d'un programme, impliquant pourtant de nombreuses manipulations de fichiers. Nos preuves, implémentations, tests et résultats sont disponibles en ligne : `https://blare-ids.org/rfblare`.

4 Conclusion

Nous avons présenté deux contributions majeures : une analyse permettant de vérifier le bon positionnement du crochet LSM dans un appel système donné et un mécanisme permettant au moniteur de flux d'information d'être résistant face aux conditions de concurrence. Ce mécanisme permet aussi de gérer une nouvelle catégorie de flux jusqu'alors hors de portée des moniteurs de flux d'information : les mémoires partagées et projections de fichiers en mémoire. Ces deux contributions visent à améliorer la confiance que l'on peut avoir dans les moniteurs de flux d'information développés pour le noyau Linux basés sur LSM.

Références

[1] Bruno DUTERTRE et Leonardo de MOURA. *The Yices SMT solver*. SRI International, 2006.

[2] Adwait NADKARNI et al. « Practical DIFC Enforcement on Android ». In : *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX : USENIX Association, août 2016, p. 1119–1136.

[3] Donald E. PORTER et al. « Practical Fine-Grained Information Flow Control Using Laminar ». In : *ACM Transactions on Programming Languages and Systems* 37.1 (nov. 2014), p. 1–51.

[4] THE COQ DEVELOPMENT TEAM. *The Coq Proof Assistant Reference Manual*. 14 déc. 2016.

[5] Chris WRIGHT et al. « Linux Security Modules : General Security Support for the Linux Kernel ». In : *USENIX Security Symposium*. San Francisco, CA, USA : USENIX Association, 2002, p. 17–31.

[6] Jacob ZIMMERMANN. « Détection d'intrusions paramétrée par la politique par contrôle de flux de références ». Thèse de doct. Université de Rennes 1, 16 déc. 2003.