Plateforme de protection de binaires configurable et dynamiquement adaptative

Kévin Le Bon, Byron Hawkins, Erven Rohou,
Univ Rennes, Inria, CNRS, IRISA
Email: first.last@inria.fr

Guillaume Hiet et Frédéric Tronel
CentraleSupelec, Inria, Univ Rennes, CNRS, IRISA
Email: first.last@centralesupelec.fr

***Résumé*—Les attaques par corruption de mémoire sont une menace sérieuse pour l'intégrité des systèmes informatiques. De nombreuses techniques ont été développées dans le but de se prémunir de ces attaques. Cependant, le déploiement de protections efficaces impose souvent une dégradation des performances du programme. Cet article propose une approche dynamique permettant d'adapter le niveau de protection du programme cible pendant son exécution en fonction du comportement observé.**

I. Introduction

La corruption de la mémoire est un problème majeur de sécurité. Ce type de faille est rendu possible notamment à cause de la gestion manuelle de la mémoire au sein de langages tels que C et C++. Beaucoup de techniques ont été développées afin de contrer cette menace [14] cependant aucune solution définitive n'a été donnée à ce jour.

Bien que certaines protections agissent statiquement sur le code du programme [9], beaucoup sont déployées durant l'exécution de celui-ci [2], [7], [8]. Ces protections sont dites dynamiques. L'intérêt de telles protections est qu'elles peuvent tenir compte du contexte d'exécution pour détecter et réagir à des attaques.

Néanmoins, déployer des protections dynamiques au sein d'un programme a un inconvénient majeur : elles dégradent les performances de la cible. Selon la protection mise en place et sa précision, le surcoût en performance peut devenir un frein sérieux à l'utilisation de la protection [14].

Dans cet article, nous proposons une approche dynamique permettant de mettre en place des protections et de les remplacer pendant l'exécution d'un programme en fonction du niveau de menace détecté. Cette approche a pour but de ne mettre en place les protections les plus lourdes que lorsqu'elles sont jugées nécessaires afin de réduire au maximum leur impact sur les performances. Notre travail, débuté en septembre 2018, se base sur la modification dynamique de binaire [6], une technique permettant de modifier un binaire durant son exécution.

II. État de l'art

Dans cette section nous établissons tout d'abord un état de l'art des solutions dynamiques développées à ce jour contre les corruptions de mémoire puis nous décrivons les limitations de ces techniques du point de vue des performances.

A. Solutions aux corruptions de mémoire

Les attaques par corruption de la mémoire peuvent être catégorisées en fonction de leur comportement [14] : corruption de code, détournement du flot de contrôle, attaque sur les données et fuite d'informations. Ces catégories d'attaques ont permis la création de politiques de sécurité, décrivant des critères qu'une protection doit remplir afin de se prémunir de ces différents types d'attaques. Dans cet article, nous porterons une attention particulière à deux de ces politiques : la *memory safety* et la *control-flow integrity (CFI)*. La *memory safety* est une politique de sécurité interdisant toute corruption de la mémoire d'un programme mais qui est en pratique très difficile à mettre en place [14]. La CFI permet de protéger le flot de contrôle d'un programme, évitant ainsi qu'un attaquant n'exécute arbitrairement du code dans le programme. La CFI est une politique de sécurité moins complète que la *memory safety* mais sa mise en place est plus simple et son impact sur les performances du programme est moindre.

1) Memory Safety: La *memory safety* est une politique de sécurité garantissant qu'il est impossible de corrompre un pointeur afin de l'exploiter. En mettant en place cette politique, aucun pointeur ne peut être modifié pendant l'exécution du programme d'une manière imprévue par le programmeur.

Certaines solutions mettent en place une *memory safety spatiale*, c'est à dire qu'elle s'assure que les pointeurs ciblent des adresses mémoires qui sont dans les bornes associées au pointeur. Par exemple, SoftBound [7] est un outil ajoutant aux pointeurs du programme cible des métadonnées. Ces métadonnées contiennent les bornes entre lesquelles l'adresse pointée doit se trouver. À chaque déréférencement d'un pointeur, un test est effectué pour vérifier que l'adresse pointée est valide.

D'autres solutions mettent en place une *memory safety temporelle*. Ces solutions s'assurent qu'un pointeur non-initialisé ou libéré ne peut pas être utilisé. Les plus connus sont probablement Memcheck de Valgrind [10] et AddressSanitizer [12]. Ces derniers détectent les bugs *use-after-free* en marquant les emplacements mémoires libérés. Ainsi, tout accès à un tel emplacement peut être facilement détecté. Ces outils utilisent des *shadow memories* afin de s'assurer qu'il est possible d'accéder à chaque octet d'une application sans risque. La *shadow memory* est ensuite instrumentée à chaque accès. À l'inverse, CETS (*Compiler Enforced Temporal Safety*) [8] marque chaque pointeur, et non les emplacements mémoire

eux-mêmes, offrant une meilleure protection [14].

Une *memory safety* complète peut être mise en place en combinant une memory safety spatiale et temporelle. Typiquement, l'addition de SoftBound et CETS permet la mise en place d'une memory safety complète. Malheureusement, la combinaison de ces deux solutions impose un trop grand surcoût en performances.

2) Control-Flow Integrity: L'intégrité du flot de contrôle (CFI) est une politique de sécurité assurant que le programme ne dévie jamais du graphe de flot de contrôle défini par le développeur. Pour ce faire, il est nécessaire de s'assurer que l'adresse cible d'un saut indirect ou l'adresse de retour de la fonction courante n'a pas été détournée par un attaquant.

Il est possible de protéger l'adresse de retour d'une fonction, stockée sur la pile, au moyen d'un canari [5] ou d'une *shadow stack* [15]. Un canari permet de détecter un dépassement de tampon en plaçant une valeur témoin entre le tampon et les emplacements mémoire à protéger. Une *shadow stack* permet de s'assurer que les adresses de retour des fonctions n'ont pas été réécrites par l'utilisateur. Une *shadow stack* apporte une meilleure protection que le canari, puisqu'elle ne protège pas que des dépassements de tampon.

Cependant, il est également nécessaire, en plus de protéger la pile, de s'assurer de l'intégrité des sauts indirects, provoqués par exemple par l'utilisation de pointeurs de fonctions ou de *vtables* en C++. En effet, on peut supposer que l'attaquant est en mesure de modifier n'importe quelle valeur dans la mémoire, à supposer que le programme ait un droit en écriture à l'emplacement de celle-ci. En toute rigueur, il faudrait aussi s'assurer de l'intégrité des sauts directs. Toutefois, on peut raisonnablement supposer, sur une plateforme moderne, que le code n'est pas modifiable, ce qui résout ce problème. On suppose typiquement que des mesures de protections telles que le W$\oplus$X sont présentes sur le système cible.

L'algorithme de Abadi [2] permet de protéger les sauts indirects en construisant des classes d'équivalence entre les différents sauts indirects du programme et en générant un identifiant unique pour chaque classe d'équivalence. Cet identifiant est inscrit en dur dans le code situé à l'adresse d'arrivée du saut. Lors de l'exécution du programme, juste avant un saut indirect, l'identifiant est écrit dans un registre puis un test d'égalité avec l'identifiant stocké à l'adresse d'arrivée détermine si le saut est valide.

Cet algorithme assure l'intégrité d'un CFG calculé statiquement, donc sans informations sur l'exécution du programme. Ce CFG représente ainsi un surensemble des sauts qu'un programme peut normalement effectuer. Il reste donc possible de détourner ce CFG via des attaques comme le *Control-Flow Bending* [4].

En conclusion, de nombreux algorithmes ont été créés afin de protéger un programme des attaques par corruption de mémoire. Cependant plus une politique de sécurité est complète et plus il est difficile de la mettre en place. En effet, les politiques les plus complètes exigent de surveiller plus de données et de comportements et les algorithmes utilisés impose des surcoûts en performances toujours plus grands. De plus, les algorithmes implémentant ces politiques n'en traitent souvent qu'un sous-ensemble.

B. Limitations actuelles

TABLE I
SURCOÛTS EN PERFORMANCES DES SOLUTIONS PROPOSÉES

Politique	Solution		Surcoût en performances
Memory Safety	Softbound + CETS	[14]	116 %
	AddressSanitizer	[12]	73 %
	Softbound	[7]	67 %
	CETS	[8]	48 %
CFI	Abadi	[2]	15 % (max 45 %)
	Shadow stack	[15]	5 %

Les protections dynamiques contre les attaques par corruption de mémoire imposent un surcoût en temps d'exécution au programme qu'elles protègent. Malheureusement, une plus grande précision ou efficacité de la protection implique un plus grand surcoût en performances [14]. Le tableau I référence les surcoûts moyen en temps d'exécution des solutions présentées précédemment.

Les protections dynamiques sont actives durant la totalité de l'exécution du programme et handicapent ce dernier y compris quand aucune attaque n'a lieu. Néanmoins, lors de l'utilisation normale du programme, les dispositifs de sécurités ne servent pas. Ainsi, le temps d'exécution dédié aux protections mises en place dans le programme est une perte de temps sèche qui n'a réellement de sens que dans une situation très particulière : celle d'une attaque. C'est la raison pour laquelle les protections les plus précises – mais aussi les plus lourdes – ne sont pratiquement jamais utilisées [14].

C. Instrumentation de binaires dynamique

L'instrumentation de binaires dynamique (DBI) consiste en l'analyse de l'exécution d'un programme via l'injection de code d'instrumentation. La DBI peut être utilisée à de nombreuses fins telles que l'optimisation des performances [3], la compilation de code à la volée (JIT) [6], le debogage ou encore à des fins de sécurisation.

L'intérêt premier de la DBI est de pouvoir modifier le code du programme cible. Il est possible ainsi de surveiller l'exécution de celui-ci en y insérant, par exemple, des *breakpoints* et des *watchpoints*.

Il est aussi possible de rajouter du code au sein du programme afin d'ajouter des protections sans nécessiter la recompilation du programme – par exemple, la mise en place d'une *shadow stack* au sein d'un programme pendant son exécution. La DBI permet donc de protéger un programme sans avoir besoin de ses sources [11].

III. APPROCHE

Dans cette section, nous expliquons plus en détail notre approche et ses particularités. Dans un premier temps, nous définissons les concepts clefs de notre plateforme de protection. Puis nous détaillons nos projets concernant l'implémentation de notre plateforme.

A. *Protection dynamique adaptative*

Notre travail porte sur une protection dynamique contre les corruptions de mémoire. La particularité de notre solution sera de s'adapter automatiquement à la situation. Notre outil mettra à disposition plusieurs approches de protection ayant différent niveau de précision et d'efficacité. Au lancement du programme seule une protection peu précise mais peu coûteuse sera déployée, le but étant de réduire au maximum le surcoût en performances lié à la détection des comportements suspects.

Si un comportement suspect est détecté, notre outil pourra déployer une protection plus lourde et plus précise et relancer l'exécution du programme où elle s'était arrêtée. Si les protections les plus lourdes détectent elles-aussi un comportement suspect, le programme sera considéré comme attaqué. Dans ce cas, des contre-mesures devront être prises, comme stopper le programme.

À terme, si les protections les plus précises ne détectent rien, elles pourront être enlevées et laisser à nouveau place à une protection plus légère, remettant l'accent sur les performances.

Ce système ne devra pas nécessiter d'intervention de la part de l'utilisateur. Ainsi, notre outil devra lui-même choisir quelle protection déployer selon la situation. Pour ce faire, il pourra employer un moteur de décision.

Afin d'activer et de désactiver des protections dans le programme, notre plateforme se basera principalement sur la DBI. Cette méthode nous permettra de protéger des binaires dont le code source n'est pas disponible ou qu'il n'est plus possible de recompiler. De plus, cette méthode évitera de modifier le programme a priori pour bénéficier des protections offertes par notre plateforme.

B. *Implémentation*

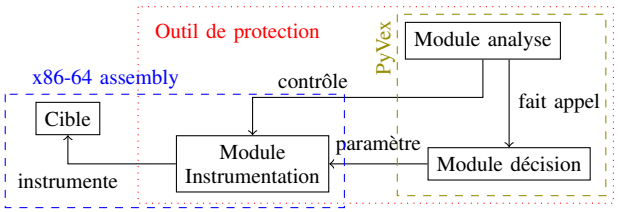

FIGURE 1. Architecture de l'outil

Notre outil prendra la forme d'un programme composé de trois modules principaux, comme le montre la figure 1. Le module d'analyse sera le module principal, il permettra de lancer des analyses sur l'état du processus cible, de faire appel au module de décisions et d'utiliser le module d'instrumentation pour interagir avec la cible. Le module de décision aura pour seule tâche de décider des protections à appliquer et retirer en fonction des résultats des analyses.

Notre module d'instrumentation utilisera la DBI pour modifier le code de la cible et pour insérer des protections. Pour ce faire, nous disposons de la bibliothèque `Padrone` [11]. Cette bibliothèque utilise l'appel système `ptrace` pour se greffer à un processus, lire et écrire des octets dans son espace d'adressage. L'intérêt d'utiliser `Padrone` pour ce projet sera que le client et la cible sont dans deux processus séparés. Ainsi, notre outil pourra effectuer de lourdes analyses sans bloquer l'exécution du programme, réduisant l'impact de notre outil sur les performances de la cible.

Dans le but de développer le module d'analyse, nous utiliserons un *framework* tel que `angr` [13]. Ce genre de *framework* nous donnera accès à un panel pré-existant d'analyses. À terme, nous pourrons éventuellement réécrire ce module en C++ afin d'améliorer les performances de notre outil.

Le module de décision pourra être écrit avec une bibliothèque telle que `PyKnow` [1]. L'intérêt de `PyKnow` est qu'il est écrit en Python, comme `angr`, ce qui facilite l'intégration de ces deux technologies au sein de notre outil.

IV. REMERCIEMENTS

Ces travaux sont partiellement financés par une bourse de thèse DGA PEC (Pôle d'Excellence Cyber).

RÉFÉRENCES

[1] PyKnow : Expert systems for Python. https://pyknow.readthedocs.io/en/stable. Accessed : 2019-01-18.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security*, 2005.

[3] Arif Ap, Kévin Le Bon, Byron Hawkins, and Erven Rohou. Fittchooser : A dynamic feedback-based fittest optimization chooser. In *16th International Conference on High Performance Computing & Simulation (HPCS 2018)-Special Session on Compiler Architecture, Design and Optimization*, page 8, 2018.

[4] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow Bending : On the Effectiveness of Control-Flow Integrity. In *USENIX Security Symposium*, pages 161–176, 2015.

[5] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, and Peat Bakke. Stackguard : Automatic adaptive detection and prevention of buffer-overflow attacks. 1998.

[6] Kim M. Hazelwood. *Dynamic Binary Modification : Tools, Techniques, and Applications*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[7] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. SoftBound : Highly compatible and complete spatial memory safety for C. *ACM Sigplan Notices*, 44(6) :245–258, 2009.

[8] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS : compiler enforced temporal safety for C. In *ACM Sigplan Notices*, volume 45, pages 31–40. ACM, 2010.

[9] George C Necula, Scott McPeak, and Westley Weimer. CCured : Type-safe retrofitting of legacy code. In *ACM SIGPLAN Notices*, volume 37, pages 128–139. ACM, 2002.

[10] Nicholas Nethercote and Julian Seward. Valgrind : a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[11] Emmanuel Riou, Erven Rohou, Philippe Clauss, Nabil Hallou, and Alain Ketterlin. PADRONE : a Platform for Online Profiling, Analysis, and Optimization. In *DCE 2014 - International workshop on Dynamic Compilation Everywhere*, Vienne, Austria, January 2014.

[12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer : A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[13] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. (State of) The Art of War : Offensive Techniques in Binary Analysis. 2016.

[14] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok : Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

[15] Stack Shield Vendicator. A stack smashing technique protection tool for Linux. http://www.angelfire.com/sk/stackshield/info.html, 2000.