

Software Fault Isolation using the CompCert compiler

Auteur: Alexandre Dang

Superviseur: Frédéric Besson
Équipe: Celtique

CentraleSupélec

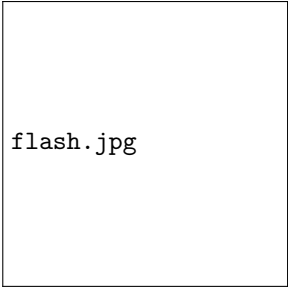
Université de Rennes 1

16 juin 2016

Flash plugin vulnérable

Connaissez-vous ce logo ?

Le plugin Flash est connu pour ces failles → conséquences sur flash
→ mais AUSSI sur votre navigateur



flash.jpg

Plan

Introduction

Fondements de Software Fault Isolation

Software Fault Isolation avec CompCert

Conclusion

Introduction

Contexte

module.pdf

- ▶ les systèmes d'exploitation avec micro-noyaux
- ▶ un navigateur web avec ses extensions

Problématique

Comment pouvoir exécuter ces modules potentiellement dangereux sans qu'ils puissent corrompre le programme principal ?

Solutions actuelles

- ▶ Isolation des modules dans différents espaces mémoires
 - ▶ Isolation par processus
 - ▶ Machines virtuelles et hyperviseur

→ les communications entre les espaces mémoires sont coûteuses en temps

- ▶ *Software Fault Isolation*

Fondements de Software Fault Isolation

Software Fault Isolation [Wahbe et al, 1993]

Définition

Software Fault Isolation permet à un programme d'exécuter des modules dans son espace mémoire de manière sécurisée.

Propriétés de sécurité de SFI

SFI garantit qu'un module respecte les propriétés suivantes :

- ▶ *Sûreté de la mémoire*, le module est confiné dans une région de la mémoire appelée *sandbox*
- ▶ *Intégrité du flot de contrôle*, les interactions extérieures à la *sandbox* sont contrôlées par une interface de confiance

Software Fault Isolation

Sandbox (bac à sable)

Espace contiguë de la mémoire où sera confiné le module à risque

- ▶ sa taille est une puissance de deux
- ▶ son adresse de départ est une puissance de deux
- ▶ identifiée par une **étiquette**

ex : 0xda est l'étiquette de la *sandbox* de la mémoire
[0xda000000 - 0xdaffffff]

Composants de SFI

- ▶ un **générateur de code**, transforme les modules afin qu'ils respectent les propriétés de SFI
→ hors de la *Trusted Computing Base*
- ▶ un **vérifieur de code**, valide que le module comporte bien les transformations du générateur
→ fait partie de la TCB

Transformations du module à risque (1/2)

- ▶ Confinement des accès mémoire :
 - ▶ *sandboxing* pour les instructions dangereuses
 - ▶ saut dans le code (`jmp`)
 - ▶ écriture dans la mémoire (`store`)

algo_sandboxing.pdf

Transformations du module à risque (2/2)

- ▶ Contrôle des appels de fonction hors de la *sandbox* via une interface de confiance faisant partie de notre TCB

`interface.pdf`

Exemple d'implémentation

NativeClient, SFI pour Google Chrome [Yee et al, 2010][Sehr and al, 2010]

- ▶ implémentation la plus aboutie de SFI
- ▶ fonctionne pour les architectures x86-32, x86-64 et ARM
 - ▶ jeu d'instructions différents
 - ▶ désassemblage du binaire plus compliqué pour le vérifieur
 - ▶ optimisations (segment mémoire physique pour x86-32, etc.)
- ▶ baisses de performances de 5% pour ARM et 7% pour x86-64.

Avantages et inconvénients

- ▶ Avantages
 - ▶ TCB réduite au vérifieur et à l'interface de contrôle des appels externes
 - ▶ approche indépendante du langage de programmation utilisée
- ▶ Inconvénients
 - ▶ le module à risque transformé est moins performant et plus lourd
 - ▶ l'implémentation de SFI dépend de l'architecture ciblée

Avantages et inconvénients

- ▶ Avantages
 - ▶ TCB réduite au vérifieur et à l'interface de contrôle des appels externes
 - ▶ approche indépendante du langage de programmation utilisée
- ▶ Inconvénients
 - ▶ le module à risque transformé est moins performant et plus lourd
 - ▶ l'implémentation de SFI dépend de l'architecture ciblée

Est-il possible d'avoir une approche de SFI portable sur plusieurs architectures ?

Software Fault Isolation avec CompCert

CompCert [Leroy, 2009]

- ▶ Compilateur certifié pour le langage C
- ▶ Écrit et prouvé avec l'assistant à la preuve Coq
- ▶ Performances proches de gcc -O1

Théorème de correction de CompCert

Tout programme S sémantiquement bien défini dans CompCert sera compilé en un code assembleur C qui aura les mêmes comportements que S

Approche SFI avec CompCert [Kroll, 2014]

Objectif : Rendre SFI portable

- ▶ transformations sur Cminor, langage indépendant de l'architecture
- ▶ transformations sémantiquement bien définies dans CompCert
- ▶ le théorème de correction de CompCert garantit que le code produit sera conforme aux exigences de SFI

compcert_pass.png

→ Le vérifieur de code n'est plus nécessaire dans l'approche SFI-CompCert

Générateur de code avec CompCert

SFI doit produire un code sécurisé quelque soit le programme en entrée

Le Cminor transformé doit :

1. respecter les propriétés de sécurité de SFI
 - ▶ opérations de *sandboxing*
 - ▶ interface de confiance pour les appels de fonction externe au module
2. être sémantiquement défini pour que le théorème de correction s'applique
 - ▶ initialisation des variables
 - ▶ vérifications complémentaires, par exemple contre la division par 0

Évaluation de l'approche

- ▶ Avantages
 - ▶ portabilité sur toutes les architectures supportées par CompCert
 - ▶ les transformations sur Cminor peuvent être optimiser durant la compilation
- ▶ Inconvénients
 - ▶ CompCert n'a pas de sémantique pour les programmes multi-tâches
 - ▶ la distribution des binaires n'est plus possible
- ▶ Performances
 - ▶ compromis entre `gcc -O0` et `gcc -O1`
 - ▶ baisse des performances de 21,7% sur x86 et 16,8% sur ARM par rapport à CompCert sans SFI

Conclusion

Conclusion

- ▶ SFI permet d'exécuter un module à risque de manière sécurisée en :
 - ▶ confinant ses accès mémoires dans la *sandbox*
 - ▶ contrôlant les appels de fonctions externes
- ▶ Deux approches possibles :
 - ▶ approche classique avec générateur de code et vérifieur de confiance
 - ▶ générateur de code avec le compilateur CompCert

Problématique du stage

- ▶ `ret` n'utilise pas de registres pour l'adresse de retour
- ▶ impossible de sécuriser par une opération de masquage dans la *sandbox*
- ▶ solution utilisée :

`ret_pop.png`

Problématique du stage

- ▶ `ret` n'utilise pas de registres pour l'adresse de retour
- ▶ impossible de sécuriser par une opération de masquage dans la *sandbox*
- ▶ solution utilisée :

`ret_pop.png`

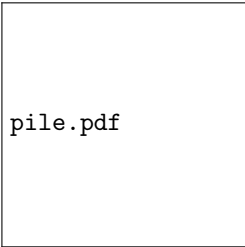
Les architectures modernes ont de nombreuses optimisations liées à l'instruction `ret`

Approche proposée

- ▶ Objectifs
 - ▶ intégrité du flot de contrôle des instructions `ret`
 - ▶ gains en performances
 - ▶ utiliser le compilateur CompCert

Approche proposée

- ▶ Objectifs
 - ▶ intégrité du flot de contrôle des instructions `ret`
 - ▶ gains en performances
 - ▶ utiliser le compilateur CompCert
- ▶ Idée
 - ▶ pile avec des trames de taille constante
 - protection des adresses de retour
 - protection contre les attaques de type *buffer overflow*



pile.pdf

Approche proposée

► Objectifs

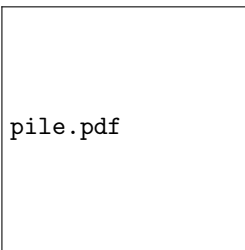
- intégrité du flot de contrôle des instructions `ret`
- gains en performances
- utiliser le compilateur CompCert

► Idée

- pile avec des trames de taille constante
 - protection des adresses de retour
 - protection contre les attaques de type *buffer overflow*

► Difficultés envisagées

- choix d'un niveau de langage pour implémenter les transformations SFI
- langage haut niveau comme Cminor, nécessite de définir une sémantique à nos transformations dans la chaîne de compilation
- langage bas niveau implémentation plus complexe



pile.pdf

Fin

Merci de votre attention

References