MSE | MASTER OF SCIENCE IN ENGINEERING

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

Master of Science HES-SO in Engineering

Orientation : Technologies de l'information et de la communication (TIC)

LEVEE, IMPLEMENTATION DE « CONTROL-FLOW INTEGRITY » AU SEIN DE LLVM

Fait par
Joël Gugger

Sous la direction de
Prof. Pascal Junod
[ResearchUnit]

Expert externe [FirstName] [LastName]
[Lab/Company]

Lausanne, HES-SO//Master, le 21 avril 2017

À propos du rapport

Information de contact

Auteur : Joël Gugger
Étudiant MSE
HES-SO//Master
Suisse
Email : *joel.gugger@master.hes-so.ch*

Declaration d'honeur

Je, soussigné, Joël Gugger, déclare que ce travail fourni est le résultat d'un travail personnel. Je certifie n'avoir usé d'aucun plagiat ou autres formes de fraudes. Toutes les ressources utilisées ainsi que les auteurs des citations ont été distinctement mentionées.

Lieu, date : _____

Signature : _____

Validation

Accepté par la HES-SO//Master (Suisse, Lausanne) sur proposition de :

Prof. Pascal Junod, conseiller du projet d'approfondissement
[FirstName] [LastName], [Lab/Company], expert principal

Lieu, date : _____

Prof. Pascal Junod	Prof. Fariba Moghaddam Bützberger
Conseiller	Resp. de la filière HES-SO//Master

Remerciements

a remplir...

Résumé

a remplir...

Mots clés : motclé1, motclé2, motclé3

Table des matières

Table des figures

Liste des tableaux

Liste des codes sources

1 Introduction

Nos programmes sont le plus souvent écrits avec des langages bas niveaux tels le C/C++ qui forcent le développeur à gérer la mémoire lui-même. Ce qui implique que, sans de bonnes connaissances et une attention particulière, un adversaire peut facilement exploiter des bugs qui surviennent au sein de ces mécanismes de gestion. Grâce à cela, l'attaquant peut modifier le "Control-Flow" de l'application et exécuter son propre code avec les privilèges donné au programme ciblé.

Sur les dix dernières années, les attaques de capables de modifier le "Control-Flow" connues au sein des principaux logiciels que nous utilisons ont augmentées. Etant donné la dangerosité de ce type d'attaque connues depuis cinquante ans, 1998 pour le "grand public", différents concepts ont été mis en place. Parmi eux on peut retrouver ASLR, DEP/NX, "Stack cookies", "Coarse-grained CFI" ou encore "Finest-grained CFI".

Mais comme à chaque fois, le jeu du chat et de la souris se met en marche et les chercheur en sécurité parviennent toujours à trouver un moyen de contourner ces mécanisme de protection. Être capable de garantir l'intergrité du "Control-Flow" de l'application est un enjeu majeur dans la sécurité des systèmes d'informations d'aujourd'hui.

C'est dans ce contexte, qu'un laboratoire de l'EPFL propose une implémentation appelée Levee qui rassemble des concepts de protection au sein de l'infrastructure de compilation LLVM. L'idée est de séparer les pointeurs jugés sensible et de les placer dans une zone mémoire sécurisée appelée "SafeStack". La séparation des pointeurs est faite par analyse durant la phase de compilation et permet d'obtenir un "hoverhead" relativement bas (environ 8% à 10%).

Le but de ce rapport est d'expliquer en détail le fonctionnement des concepts de protection de Levee ainsi que d'expérimenter et d'analyser son implémentation. Cependant, pour mieux comprendre les enjeux se cachant derrière les concepts de Levee un bref récapitulatif du fonctionnement de la mémoire au sein des systèmes d'exploitations modernes ainsi qu'un historique des protections et leurs attaques respectives est dressé dans le chapitre suivant.

2 Historique des mécanismes de protection

La gestion de la mémoire est un des composants le plus complexe d'un système d'exploitation moderne. Ce qui rend le sujet bien plus vaste que ce que l'on peut traiter dans ce rapport. Cependant, il m'a été nécessaire de parcourir les principaux concepts pour pouvoir en comprendre les enjeux.

Dans ce chapitre un bref récapitulatif de cette gestion est faite en préambule de la partie historique des attaques et des mécanismes de protection. Les cas expliqués dans ce rapport sont volontairement simplifiés de manière à comprendre l'aspect conceptuel et non pratique. Exploiter dans un environement réel certaines des attaques brievement décritent par la suite peut occuper la place d'un rapport au moins égal à celui-ci.

Contenu

2.1 Rappel sur la gestion de la mémoire

La mémoire d'un programe est gérée selon un schéma bien définit. Chaque processus du système d'exploitation voit sa mémoire définie dans un bac-à-sable appelé "virtual address space". Ce espace est toujours égal à 4Go dans un système 32 bits. Le système d'exploitation est ensuite responsable de faire le lien entre cet espace mémoire vituel et l'épace d'adresses physique.

Cette mémoire virtuelle est d'abord scindée en deux parties. Cependant cela ne signifie pas que l'espace est entièrement utilisé. La première ayant les adresses mémoires 0xc0000000 à 0xffffffff est reservée au noyaux du système d'exploitation sous linux. La seconde correspond à l'espace disponible au programme.

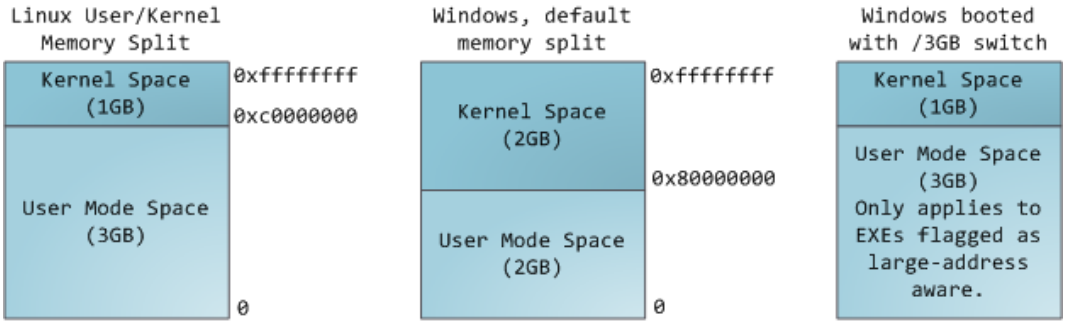

FIGURE 2.1 *Répartition de l'espace mémoire entre le noyau et le programme, par G. Duarte*
Source: *http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/*

L'espace réserver au programme est ensuite découpé en différents segments tel que la Stack, Heap, etc. Ces segments sont des plages mémoires continues gérée par le système d'exploitation. Dans le cas d'un processus Linux les ségments sont répartit ainsi :

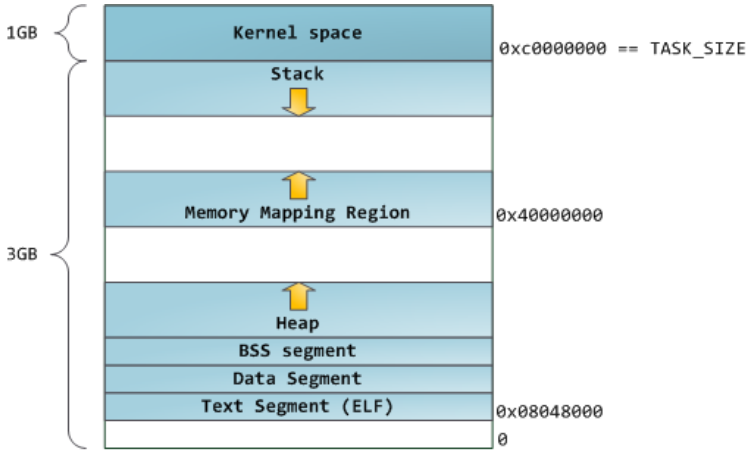

FIGURE 2.2 *Ségmentation de la mémoire d'un processus Linux 32 bits, par G. Duarte*
Source: *http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/*

La Stack permet de gérer le "Control-Flow" de l'application. À chaque appel de fonction, une nouvelle stack frame est ajoutée à la Stack et est ensuite retirée lorsque celle-ci se termine. La Stack grandit vers le bas, c-à-d que les adresses mémoires sont décroissantes. Il est possible que la Stack veuille s'étende au-delà de sa taille maximum, c'est le stack overflow et dans ce cas le programme recoit un "segmentation fault".

Le segment "Memory Mapping Region" permet au noyaux de copier en mémoire le contenu de certain fichier de manière à augmenter les performances. Ce segment est généralement utilisé pour charger les librairies. Il peut aussi être utilisé à d'autre fin, à la place de la Heap par exemple.

En dessous se trouve la Heap, permettant de stocker en mémoire les allocations dynamiques. En C ce ségment est géré par la fonction `malloc()` ainsi que ses collèges. Dans d'autres langages bénéficiant d'un ramasse mièttes tel que le C#, l'interface pour intéragir avec la Heap est le mot reservé `new`.

Finalement les trois derniers segements que sont BSS, Data et Text servent a stocker les variables static initialisées ou non ainsi que la source du binaire executé. En Figure 2.3 un exemple de ce que l'on peut retrouver dans ces trois segments :

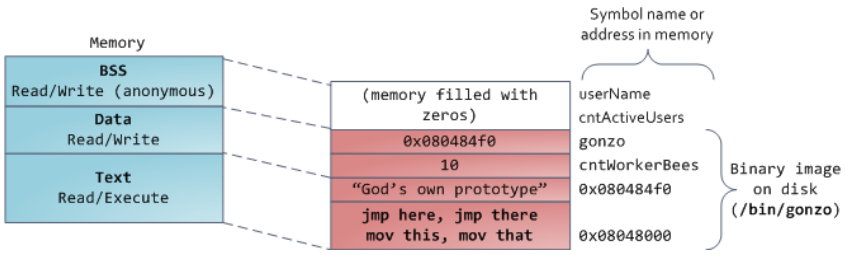

FIGURE *2.3* *Mapping d'une image binaire dans les ségments BSS, Data et Text, par G. Duarte*
Source: *http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/*

Lors de l'execution d'un programme, cette mémoire virtuelle est gérée par le système d'exploitation grâce à une structure appelée "Memory Descriptor". Cette structure contient les adresses de début et de fin de chaque ségments.

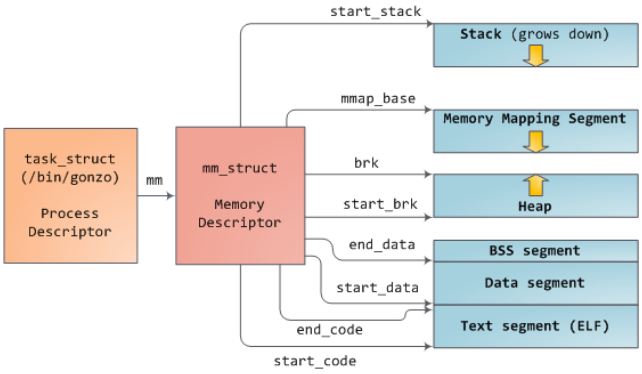

FIGURE *2.4* *Memory Descriptor d'un processus Linux, par G. Duarte*
Source: *http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/*

Cette structure est constituée d'une suite de vm_area_struct. Chacun d'eux est un espace continu en mémoire. Ils permettent de stocker des informations tels que les droits d'écriture et de lecture ou encore les droits d'execution. Ils stockent aussi si et quel fichier est copier en mémoire.

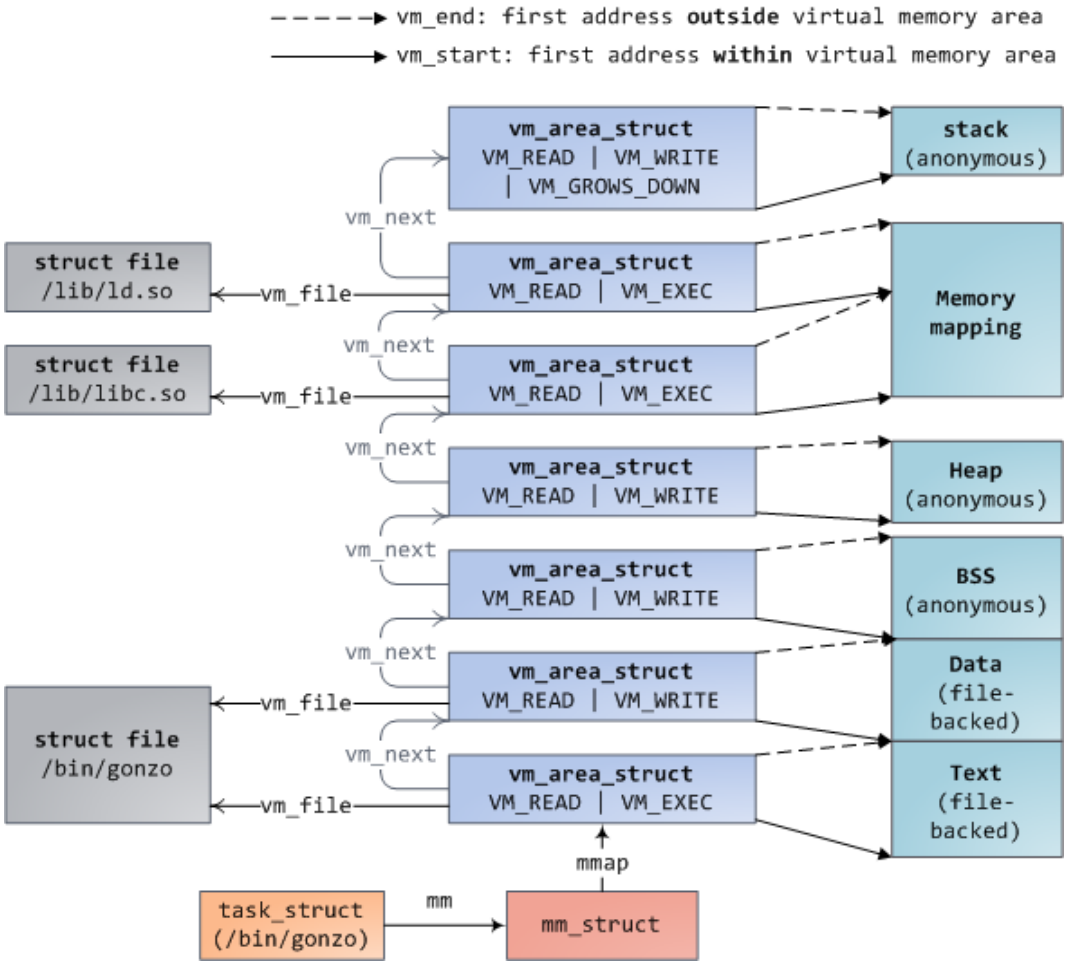

FIGURE *2.5* *Structure des "Virtual Memory Area", par G. Duarte*
Source: *http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/*

2.2 Le buffer overflow

Le buffer overflow consiste à utiliser une fonction qui ne vérifie par la taille de l'argument à copier en mémoire, par exemple `strcpy()`, pour redéfinir l'adresse de retour de la fonction et modifier le flow de l'application en le redirigeant à un endroit ou l'attaquant aura, par exemple, péalablement injecter son code (p.ex. un shell code).

Lorsqu'une Stack frame est créée, celle-ci stocke dans un schéma particulier les informations d'ont elle a besoin.

1. les paramètres passé à la fonction
2. l'adresse de retour
3. une sauvegarde du pointeur %ebp

4. et les variables locales

Cela permet, en dépassant la taille des variables locales, de modifier des zones mémoires qui ne devraient pas l'être. En regardant la Figure 2.6 on constat que si l'on écrit (8+4+4+4) = 12 bytes dans le local_buffer, les 4 dernier bytes auront remplacé l'adresse de retour.

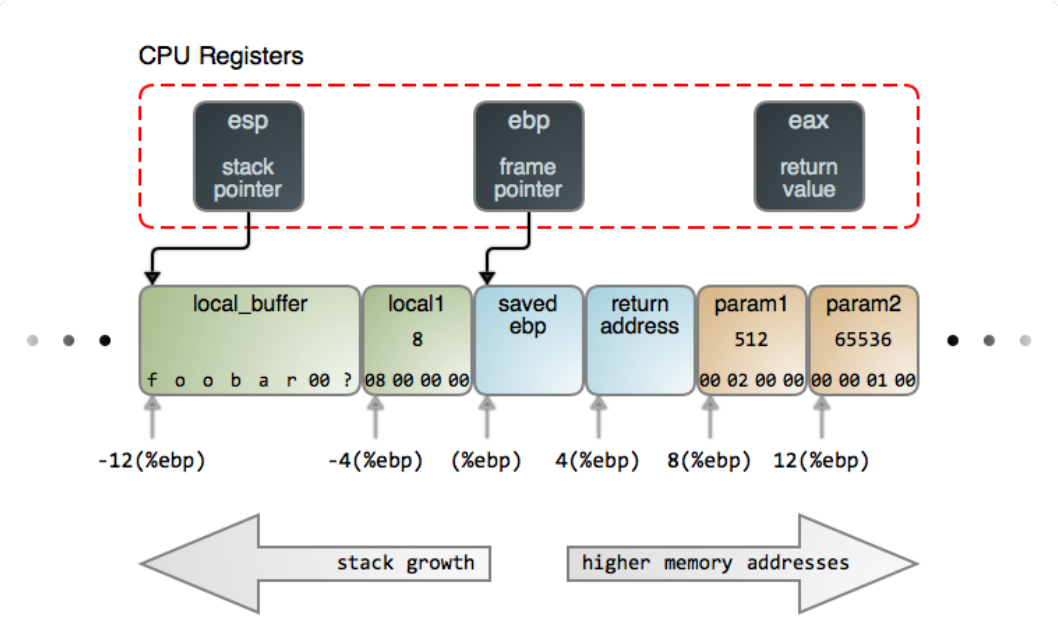

FIGURE 2.6 *Exemple d'une Stack frame*
Source: *http://duartes.org/gustavo/blog/post/journey-to-the-stack/*

Pour illustrer ce cas, voici un code C utilisant la fonction `strcpy()`. Dans cette exemple trivial il est possiblde d'injecter un shell code dans le buffer et de redefinir l'adresse de retour. La chaine de caractère copiée dans le local_buffer est directement contrôlée par l'utilisateur, ce qui rend la manoeuvre encore plus facile.

```
1  #include <stdlib.h>
2
3  void func(char *param1)
4  {
5    char local_buffer[100];
6    strcpy(local_buffer, param1);
7  }
8
9  int main(int argc, char **argv)
10 {
11   func(argv[1]);
12   return 0;
13 }
```

Listing 2.1 *Exemple de programe vulnérable au buffer overflow*

2.3 DEP/NX

Pour éviter lors d'un buffer overflow que l'attaquant puisse executer du code aux endroits mémoire censés contenir des données, les zones mémoires doivent être marquées comme étant executable ou non-executable. Cette information est justement stockée dans la structure "Virtual Memory Area" sour Linux.

2.3.1 Mécanisme de protection

DEP pour Data Execution Prevention a été introduit sur Linux en 2004 avec la version 2.6.8. [1]

La protection se base sur le hardware, le NX bit, introduit tout d'abord par AMD en 2003, puis reprise par Intel sous le nom de XD bit une année après. [2] Ce bit indique au processeur s'il sagit d'une zone d'instructions ou de données. Cependant cette fonctionalité hardware peut être simulée, entrainant du coup un overhead important.

2.3.2 Contournements avec return-to-libc

Une pile non-executable ne permet plus à l'attaquant d'executer son code, mais cela n'empêche pas d'executer du code executable déjà présent dans le programe. Comme montré dans la Figure 2.5, la bibliothèque partagée libc est présente. Ce qui rend possible une attaque de type return-to-libc. Grâce à la fonction `system()` il est possible d'executer arbitrairement un programe. Lors de l'attaque on localise par exemple une chaine de caractère tel que `"/bin/sh"` que l'on prepare comme étant le paramètre à passé à la fonction `system()`.

2.4 ASLR (Address space layout randomization)

Comme montré sur la Figure 2.3, l'espace d'adressage virtuel est structuré de manière fixe. De cette manière il est possible de prévoir ou se trouve en mémoir les différents composant de notre programe. L'attaque de type return-to-libc à besoin de connaitre l'adresse de la fonction `system()` et de la chaine de caractère `"/bin/sh"`. Dans le cas ou ces adresses changes à chaque lancement, la tâche devient plus compliquée.

1. Dans la même année pour Windows et deux ans plus tard pour Mac OS X lors de la transition vers x86 en 2006. Source : `http://duartes.org/gustavo/blog/post/journey-to-the-stack/`
2. Source : `https://fr.wikipedia.org/wiki/NX_Bit`

2.4.1 Mécanisme de protection

Depuis juin 2005, l'Address Space Layout Randomization est supportée dans le noyau Linux avec la version 2.6.12. Afin de rendre imprédictible les adresses sensibles, trois décalages aléatoires sont effectués au sein de la mémoire virtuelle. Le premier permet de décaler la Stack vers le bas, le second décale lui aussi vers le bas le segment de Mapping et le dernier décale vers le haut le segment de la Heap.

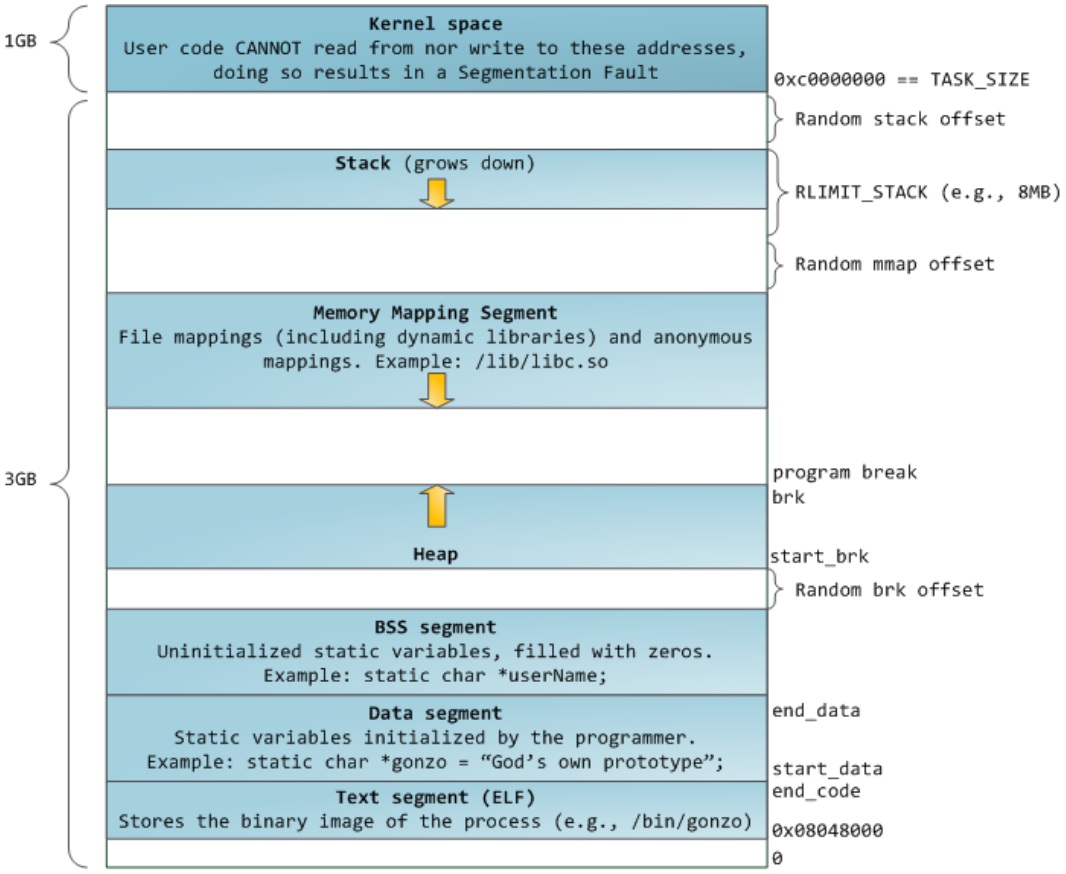

FIGURE *2.7* *Address space layout randomization*
Source: *http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/*

2.4.2 Limitation et contournements

Sur un OS 32 bits, comme montré a la Figure 2.7, la marge de manoeuvre laissée au décalage n'est pas trés grande. Seul une partie des bits est utilisée, ce qui laisse possible une attaque de type force brutte réussir en quelques milliers d'essais. En effet la Stack est randomisée avec une entropie de 19 bits et la Memory Mapping 8 bits.

Dans le cas d'un OS 64 bits ASLR devient tout de suite plus interessant car l'espace mémoire virtuel est beaucoup plus grand. Cependant, les chercheurs Hector Marco-Gisbert et Ismael Ripoll de l'université de Valence ont écrit un papier démontrant une faiblesse sous certaines assomptions.

2.5 Les stack cannaries

2.6 Control-Flow integrity

3 | Analyse de Levee

introduction du projet Levee... composant, équipe, epfl

Contenu

3.1 Concepts théoriques

introduction au concept de separation des pointeurs

3.1.1 CPI/CPS

Decrire pourquoi les pointeurs sensibles et pas sensibles. comment determiner si un pointeur est sensible

3.1.2 SafeStack

quel est le concept de la safeStack

3.2 Implémentation au sein de LLVM

version de LLVM, depuis quand, sous quel nom, documentation

structure de LLVM front-end, l'optimizer, et le back-end, son fonctionnement, origine

3.2.1 Structure

description des actions effectuée dans le front-end, l'optimizer, et le back-end

3.3 Rayon d'action

qu'est qu'il est sensé proteger par rapport au chapitre historique

4 Proof of Concept d'une attaque

SafeStack doit normalement prévenir les attaques de types XXX. Dans ce chapitre un proof of concept d'une telle attaque est décrit ainsi que les moyens mis en oeuvre par SafeStack pour la bloquée.

Contenu

4.1 Contexte

Environement dans lequel se passe l'attaque
Description du docker
Quels mécanisme sont actifs ou non

4.2 Description théorique de l'attaque

Description des étapes de l'attaque et des réaction attendue

4.3 Implémentation

On essaie de le faire / just do it

5 Conclusions

5.1 Les innovations apportées par Levee

Y a-t-il des innovations et lequels

5.2 Évaluation des objectifs initiaux

rempli, pas rempli...

5.3 Difficultés rencontrées

5.4 Sujet de recherche à développer

A An appendix

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Donec odio elit, dictum in, hendrerit sit amet, egestas sed, leo. Praesent feugiat sapien aliquet odio. Integer vitae justo. Aliquam vestibulum fringilla lorem. Sed neque lectus, consectetuer at, consectetuer sed, eleifend ac, lectus. Nulla facilisi. Pellentesque eget lectus. Proin eu metus. Sed porttitor. In hac habitasse platea dictumst. Suspendisse eu lectus. Ut mi mi, lacinia sit amet, placerat et, mollis vitae, dui. Sed ante tellus, tristique ut, iaculis eu, malesuada ac, dui. Mauris nibh leo, facilisis non, adipiscing quis, ultrices a, dui.