



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 23 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'honneur

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

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
Conseiller

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

Remerciements

a remplir...

Résumé

a remplir...

Mots clés : motclé1, motclé2, motclé3

Table des matières

Remerciements	v
Résumé	vii
Table des figures	xi
Liste des tableaux	xiii
Liste des codes sources	xv
1 Introduction	1
2 Historique des mécanismes de protection	3
2.1 Rappel sur la gestion de la mémoire	4
2.2 Le buffer overflow	6
2.3 DEP/NX	8
2.4 ASLR (Address space layout randomization)	8
2.5 Les stack canaries	10
2.6 Control-Flow integrity	10
3 Analyse de Levee	11
3.1 Concepts théoriques	12
3.2 Implémentation au sein de LLVM	12
3.3 Rayon d'action	12
4 Proof of Concept d'une attaque	13
4.1 Contexte	14
4.2 Description théorique de l'attaque	14
4.3 Implémentation	14
5 Conclusions	15
5.1 Les innovations apportées par Levee	15
5.2 Évaluation des objectifs initiaux	15
5.3 Difficultés rencontrées	15
5.4 Sujet de recherche à développer	15
A An appendix	17
Références	21

Table des figures

2.1	Répartition de l'espace mémoire du kernel	4
2.2	Ségmentation de la mémoire d'un processus Linux 32 bits	4
2.3	Mapping d'une image binaire dans les segments BSS, Data et Text . . .	5
2.4	Memory Descriptor d'un processus Linux	5
2.5	Structure des "Virtual Memory Area"	6
2.6	Exemple d'une Stack frame	7
2.7	Address space layout randomization	9

Liste des tableaux

Liste des codes sources

2.1	Exemple de programme vulnérable au buffer overflow	7
-----	--	---

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'intégrité 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 environnement réel certaines des attaques brièvement décrites par la suite peut occuper la place d'un rapport au moins égal à celui-ci.

La description du fonctionnement de la mémoire est très fortement basée sur les articles suivants [1] [2] [3] tirés du blog de Gustavo Duarte.

Contenu

2.1	Rappel sur la gestion de la mémoire	4
2.2	Le buffer overflow	6
2.3	DEP/NX	8
2.3.1	Mécanisme de protection	8
2.3.2	Contournements avec return-to-libc	8
2.4	ASLR (Address space layout randomization)	8
2.4.1	Mécanisme de protection	9
2.4.2	Limitation et contournements	9
2.5	Les stack canaries	10
2.6	Control-Flow integrity	10

2.1 Rappel sur la gestion de la mémoire

La mémoire d'un programme est gérée selon un schéma bien défini. 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 4 Go dans un système 32 bits. Le système d'exploitation est ensuite responsable de faire le lien entre cet espace mémoire virtuel et l'espace 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 réservée au noyau du système d'exploitation sous linux. La seconde correspond à l'espace disponible au programme.

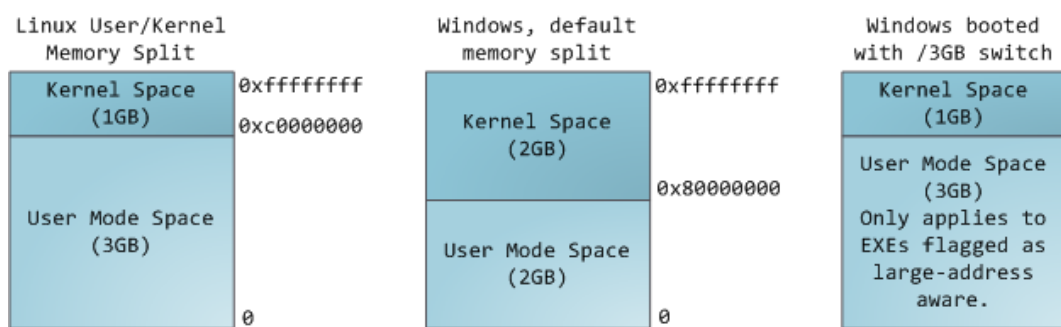


FIGURE 2.1 Répartition de l'espace mémoire entre le noyau et le programme, par G. Duarte
Source: <http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/>

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

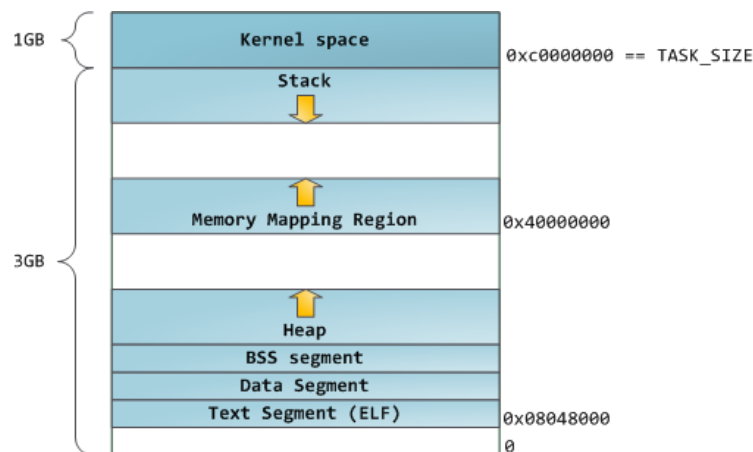


FIGURE 2.2 Segmentation de la mémoire d'un processus Linux 32 bits, par G. Duarte
Source: <http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/>

2.1. Rappel sur la gestion de la mémoire

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'étendre au-delà de sa taille maximum, c'est le stack overflow et dans ce cas le programme reçoit un "segmentation fault".

Le segment "Memory Mapping Region" permet au noyau de copier en mémoire le contenu de certain fichier de manière à augmenter les performances. Ce segment est généralement utilisé pour charger les bibliothèques. 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 segment est géré par la fonction `malloc()` ainsi que ses collègues. Dans d'autres langages bénéficiant d'un ramasse-miettes tel que le C#, l'interface pour interagir avec la Heap est le mot réservé `new`.

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

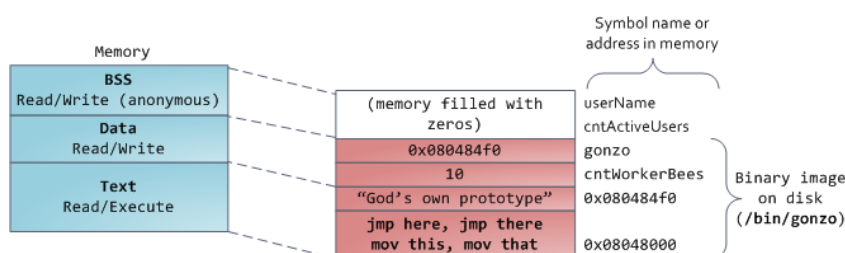


FIGURE 2.3 Mapping d'une image binaire dans les segments BSS, Data et Text, par G. Duarte
Source: <http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/>

Lors de l'exécution 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 segments.

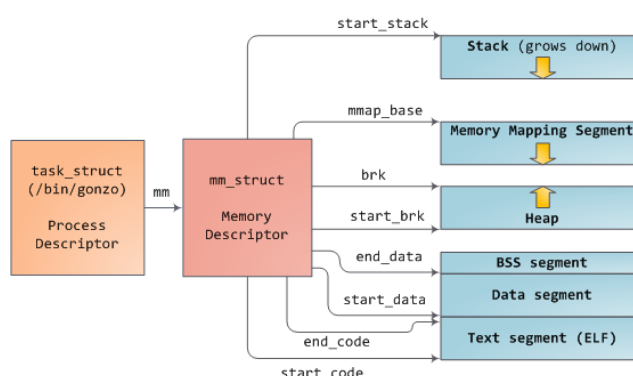


FIGURE 2.4 Memory Descriptor d'un processus Linux, par G. Duarte
Source: <http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/>

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'exécution. Ils stockent aussi si et quel fichier est copier en mémoire.

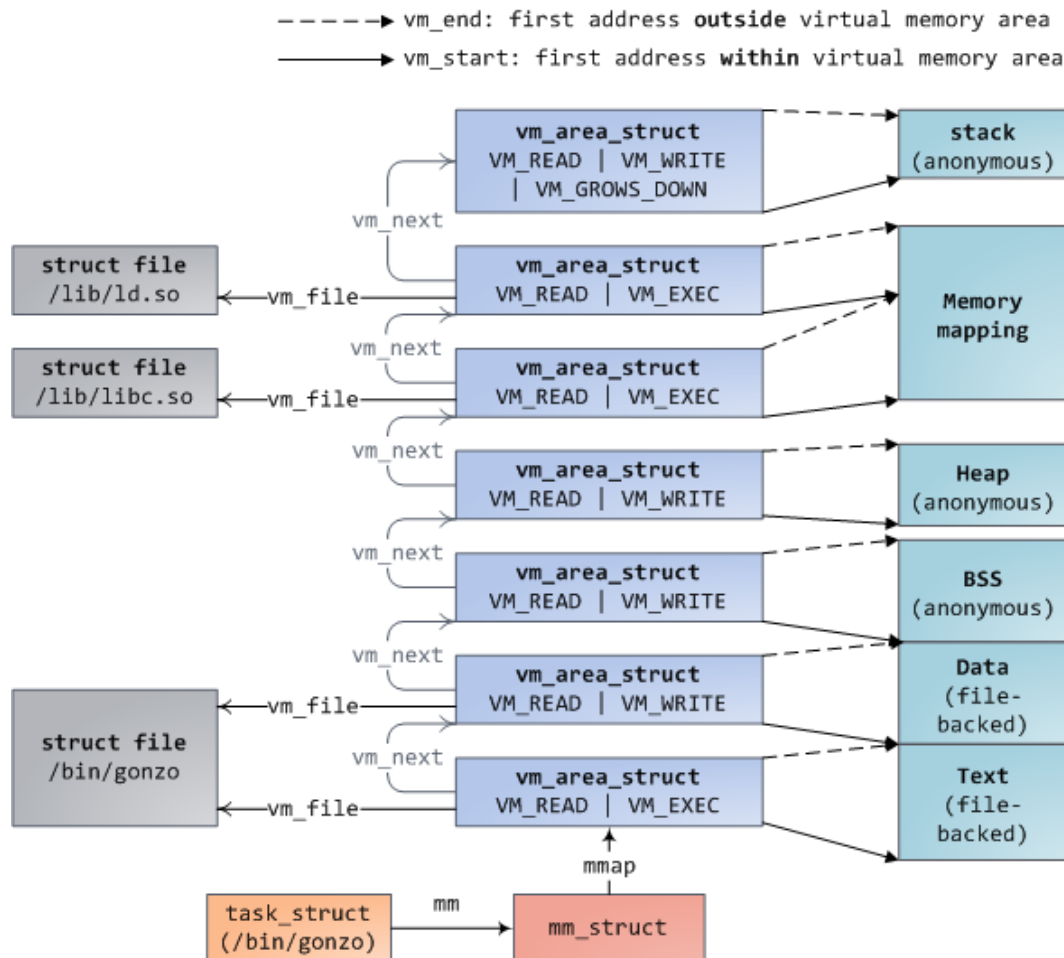


FIGURE 2.5 Structure des "Virtual Memory Area", par G. Duarte
 Source: <http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/>

2.2 Le buffer overflow

Le buffer overflow consiste à utiliser une fonction qui ne vérifie pas 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 où 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 dont elle a besoin.

1. les paramètres passés 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 $(8+4+4+4) = 20$ bytes dans le `local_buffer`, les 4 dernier bytes auront remplacé l'adresse de retour.

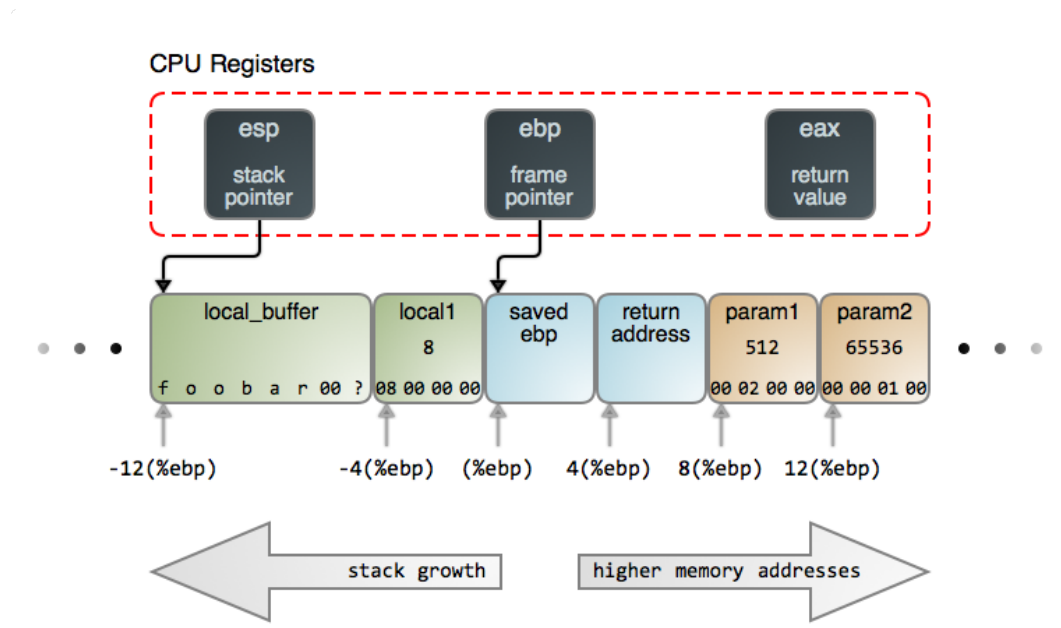


FIGURE 2.6 Exemple d'une Stack frame

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 possible d'injecter un shell code dans le buffer et de redéfinir l'adresse de retour. La chaîne 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 programme vulnérable au buffer overflow

2.3 DEP/NX

Pour éviter lors d'un buffer overflow que l'attaquant puisse exécuter du code aux endroits mémoire censés contenir des données, les zones mémoires doivent être marquées comme étant exécutable ou non-exécutable. Cette information est justement stockée dans la structure "Virtual Memory Area" sous 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 du noyau, durant la même année pour Windows et deux ans plus tard pour Mac OS X lors de la transition vers x86 en 2006 [4].

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 [5] [6]. Ce bit indique au processeur s'il s'agit d'une zone d'instructions ou de données. Cependant cette fonctionnalité hardware peut être simulée, entraînant du coup un overhead important.

2.3.2 Contournements avec return-to-libc

Une pile non-exécutable ne permet plus à l'attaquant d'exécuter son code, mais cela n'empêche pas d'exécuter du code exécutable déjà présent dans le programme. Comme montré dans la Figure 2.5, la bibliothèque partagée libc est présente. Ce qui rend possible une attaque de type return-to-libc[7]. Grâce à la fonction `system()` il est possible d'exécuter arbitrairement un programme. Lors de l'attaque on localise par exemple une chaîne de caractère tel que `"/bin/sh"` que l'on prépare comme étant le paramètre à passer 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 où se trouve en mémoire les différents composants de notre programme. L'attaque de type return-to-libc a besoin de connaître l'adresse de la fonction `system()` et de la chaîne de caractère `"/bin/sh"`. Dans le cas où ces adresses changent à chaque lancement, la tâche devient plus compliquée.

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.

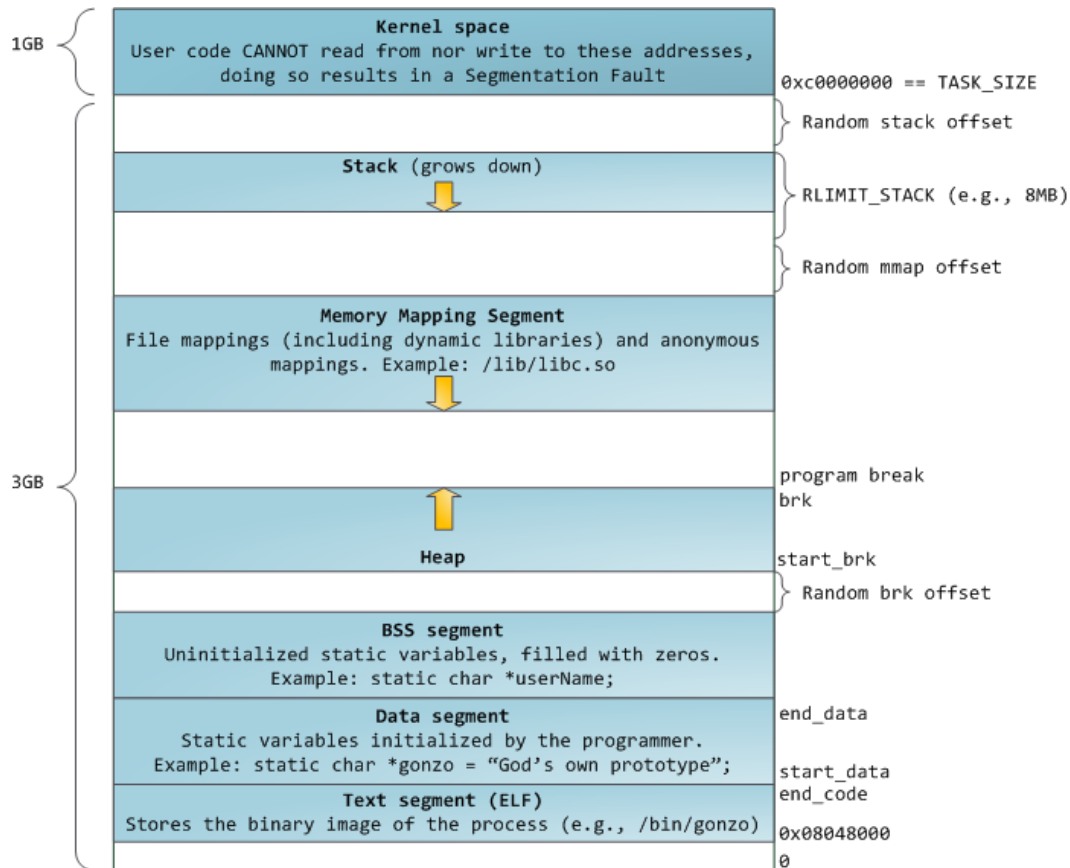


FIGURE 2.7 Address space layout randomization

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 à la Figure 2.7, la marge de manoeuvre laissée au décalage n'est pas très grande. Seul une partie des bits est utilisée, ce qui laisse possible une attaque de type force brute réussie 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 intéressant 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 assumptions [8].

2.5 Les stack canaries

2.6 Control-Flow integrity

3 | Analyse de Levee

Levee est un projet mener dans le cadre du Dependable Systems Lab [9] par les chercheurs Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar et Dawn Song.

Le but annoncé du projet est de sécurisé tout programme informatique contre la totalité des attaques de type "control-flow hijack" via une erreur mémoire. Comme montré dans le chapitre précédant, il existe déjà quelques mécanismes (DEP, ASLR) permettant de réduire le risque de ce type d'attaque sans imposer un cout supplémentaire à la performance du programme. Cependant il est possible de les contourner (return-to-libc, ROP). D'autres mécanismes (CFI) permettent quant à eux d'améliorer fortement la sécurité, mais ne sont pas adoptés majoritairement pour cause de leur cout élevé en performance.

Toutes ces techniques ne permettent pas de garantir l'intégrité complète du control-flow sans imposer de cout élevé ou sans demander au programmeur de modifier le code source de son programme. Dans le cas des langages de type "memory-safe", un objet en mémoire ne peut être accédé que depuis un pointeur prévu explicitement pour l'objet en question. Cela rend la modification du control-flow impossible mais entraîne une baisse de performance importante.

Dans ce chapitre seront abordés les concepts théoriques sur lesquels se base Levee ainsi que son implémentation au sein du compilateur LLVM.

Contenu

3.1 Concepts théoriques	12
3.1.1 CPI (Code-pointer integrity)	12
3.1.2 CPS (Code-pointer separation)	12
3.1.3 Safe Stack	12
3.2 Implémentation au sein de LLVM	12
3.2.1 Structure	12
3.3 Rayon d'action	12

3.1 Concepts théoriques

Les chercheurs du projet posent comme postulat de départ qu'il est suffisant de garantir l'intégrité des pointeurs pour rendre impossible la modification du control-flow par exploitation d'erreurs mémoire.

Afin de garder de bonne performance tout en garantissant leur intégrité, le code est analysé de manière statique à la compilation. Le concept de CPI [10], pour "code-pointer integrity", intervient alors afin de déterminer quels pointeurs doivent être protégés.

Seul les attaques visant à dérouté le control-flow sont prise en compte dans leur modèle de sécurité. Les attaques de type "data-only", visant à modifier ou récupérer des informations qui ne font pas partie du control-flow n'entrent pas en considération.

Ils assument le fait que l'attaquant a le contrôle total sur la mémoire du processus et que le chargement du programme ainsi que le binaire ne peuvent pas être altérés. De ce fait, l'instrumentation du programme résultant de la compilation peut se mettre en place avant intervention de l'attaquant.

3.1.1 CPI (Code-pointer integrity)

Décrire pourquoi les pointeurs sensibles et pas sensibles. comment déterminer si un pointeur est sensible

3.1.2 CPS (Code-pointer separation)

Décrire la variante CPS, moins d'overhead mais permettant certain hijack

3.1.3 Safe Stack

quel est le concept de la safe stack

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 protéger 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	14
4.2	Description théorique de l'attaque	14
4.3	Implémentation	14

4.1 Contexte

Environnement 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 lesquels

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 consectetur.

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, consectetur at, consectetur 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.

fdsf

Références

- [1] Gustavo DUARTE. *Anatomy of a Program in Memory*. URL : <http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/> (visité le 27/01/2009).
- [2] Gustavo DUARTE. *How the Kernel Manages Your Memory*. URL : <http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/> (visité le 03/02/2009).
- [3] Gustavo DUARTE. *Journey to the Stack, Part I*. URL : <http://duartes.org/gustavo/blog/post/journey-to-the-stack/> (visité le 10/03/2014).
- [4] *Data Execution Prevention*. URL : https://fr.wikipedia.org/wiki/Data_Execution_Prevention (visité le 09/06/2015).
- [5] *Executable space protection*. URL : https://en.wikipedia.org/wiki/Executable_space_protection (visité le 08/01/2017).
- [6] *NX Bit*. URL : https://fr.wikipedia.org/wiki/NX_Bit (visité le 21/03/2017).
- [7] *Return-to-libc attack*. URL : https://fr.wikipedia.org/wiki/Return-to-libc_attack (visité le 07/06/2016).
- [8] Hector MARCO-GISBERT et Ismael RIPOLL. *On the Effectiveness of Full-ASLR on 64-bit Linux*. Rapp. tech. Universitat Politècnica de València, nov. 2014. URL : <http://cybersecurity.upv.es/attacks/offset2lib/offset2lib-paper.pdf>.
- [9] *Dependable Systems Lab*. URL : <http://dslab.epfl.ch/> (visité le 23/04/2017).
- [10] Volodymyr KUZNETSOV et al. *Code-Pointer Integrity*. Rapp. tech. Ecole Polytechnique Fédérale de Lausanne (EPFL), UC Berkeley, Stony Brook University, Purdue University, oct. 2014. URL : <http://dslab.epfl.ch/pubs/cpi.pdf>.

