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 mai 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. Cela 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 flot de contrôle de l'application et exécuter son propre code avec les privilèges donnés au programme ciblé.

Sur les dix dernières années, les attaques de capables de modifier le flot de contrôle 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 » [1, 2]), les universités ainsi que les chercheurs en sécurité informatique des grands groupes (IBM, Intel, Google, Microsoft, etc) ont proposés et mis en place différents concepts de protection visant à empêcher ce type spécifique d'attaques. Parmi ces mécanismes de protection, on retrouve ASLR, DEP/NX, « Stack cookies » ou encore différentes implémentations de « Control-Flow integrity (CFI) » telles que « Coarse-grained CFI » et « Finest-grained CFI ».

Mais comme à chaque fois, le jeu du chat et de la souris se met en marche et d'autres chercheurs en sécurité parviennent toujours à trouver un moyen de contourner ces mécanismes de protection. Être capable de garantir l'integrité du flot de contrôle 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'École polytechnique fédérale de Lausanne (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 sensibles et de les placer dans une zone mémoire particulière. La séparation des pointeurs est faite par analyse durant la phase de compilation et permet d'obtenir un coût en performance relativement bas (environ 8% à 10%).

Le but de ce rapport est d'expliquer en détail le fonctionnement des concepts de protection sur lequels Levee se base et d'expérimenter et analyser son implémentation. Pour mieux comprendre les enjeux se cachant derrière ces concepts, un bref récapitulatif du fonctionnement de la mémoire au sein des systèmes d'exploitations modernes ainsi qu'un historique des mécanismes de protection et des attaques possibles 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 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 inspirée des articles suivants [3, 4, 5] tirés du blog de Gustavo Duarte. À des fins de simplicité, les concepts exposés sont basé sur une architecture 32 bits. Dans le cas de changements notables entre architectures, un complément spécifique en 64 bits est donné. Les termes anglais sont mentionnés une première fois lors de leurs traductions, le document utilisera ensuite la traduction française.

Contenu du chapitre

2.1 Rappel sur la gestion de la mémoire

La mémoire d'un programe est gérée selon un schéma bien défini. Chaque processus du système d'exploitation voit sa mémoire définie dans un espace virtuel « Virtual address space », l'isolant complètement du reste des processus. Ce espace est égal à 4 Go dans un système 32 bits. Dans le cas d'une architecture 64 bits, l'espace disponible n'utilise pas 2^{64} bytes (16 Eo), mais uniquement les 48 bits les moins significatifs, et donc 2^{48} bytes (256 To) [6, 7]. Le système d'exploitation est ensuite responsable de faire le lien entre cet espace virtuel et l'espace d'adressage physique.

2.1.1 Segmentation

La mémoire virtuelle est scindée en deux parties principales. La première, ayant les adresses mémoires allant de `0xc0000000` à `0xffffffff` (en 32 bits), est reservée, sous Linux, au noyau du système d'exploitation. La seconde, quant à elle, correspond à l'espace disponible du processus courant.

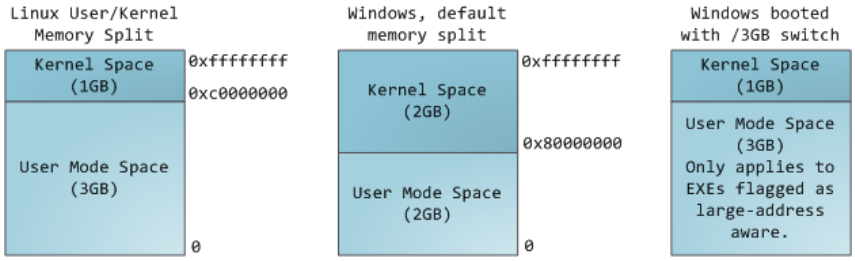

FIGURE 2.1 *Répartition de l'espace mémoire virtuel entre le noyau et le programme, par G. Duarte*
Source: *http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/*

L'espace réservé au processus est ensuite découpé en différents segments tel que la pile « Stack » ou le tas « Heap ». 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 :

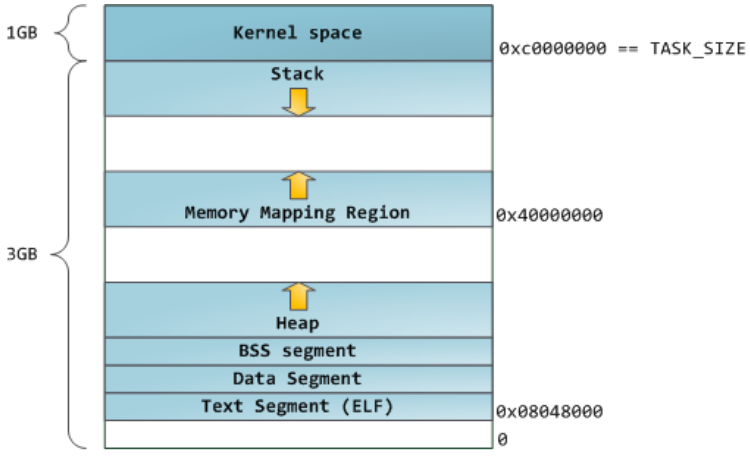

FIGURE 2.2 *Segmentation de la mémoire d'un processus Linux en 32 bits, par G. Duarte*
Source: *http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/*

La pile d'exécution permet de gérer le flot de contrôle de l'application. À chaque appel de fonction, une nouvelle structure de pile « Stack frame » est ajoutée à la pile, puis est retirée lorsque la fonction se termine. La pile d'exécution grandit vers le bas, c'est-à-dire que les adresses mémoires décroissent lorsque la pile se remplie. Il est possible que la pile veuille s'étende au-delà de sa taille maximum, c'est le cas du dépassement de pile « Stack overflow ». Dans ce cas, le programme reçoit une erreur de segmentation « Segmentation fault » et s'arrête.

Le segment « Memory Mapping Region » permet au noyau de copier en mémoire le contenu de certains fichiers de manière à augmenter les performances. Ce segment est généralement utilisé pour charger les librairies dynamiques. Il peut aussi être utilisé à d'autres fins, par exemple à la place d'utiliser le tas pour stocker certaines données.

En dessous se trouve le tas, permettant de stocker en mémoire les allocations dynamiques. En C, ce segment est géré par la fonction `malloc()` et ses confrères. Dans d'autres langages bénéficiant d'un ramasse miettes, tel que le C#, l'interface pour intéragir avec le tas est le mot réservé `new`.

Finalement les trois derniers segments que sont « BSS », « Data » et « Text » servent à stocker les variables statiques 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 segments :

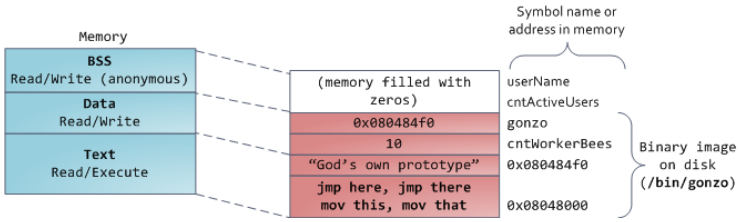

FIGURE 2.3 *Plan d'une image binaire dans les segments BSS, Data et Text, par G. Duarte*
Source: *http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/*

2.1.2 Descripteurs mémoires

Lors de l'exécution d'un programme, cet espace mémoire est géré par le système d'exploitation grâce à des descripteurs de mémoire appelés « Memory Descriptor ». Cette structure contient les adresses de début et de fin de chaque segments.

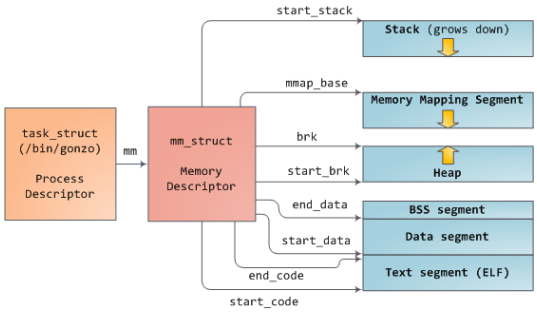

FIGURE 2.4 *Descripteur de mémoire d'un processus Linux, par G. Duarte*
Source: *http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/*

Cette structure globale est constituée d'une suite de plus petites structures appelées espaces virtuels de mémoire « Virtual Memory Area », `vm_area_struct` sur la Figure 2.5. Chacune d'elles est un espace continu en mémoire et permettent de stocker des informations tels que les droits d'écriture, de lecture ou encore les droits d'exécution du contenu mémoire. Elles stockent également si et quel fichier est mappé en mémoire.

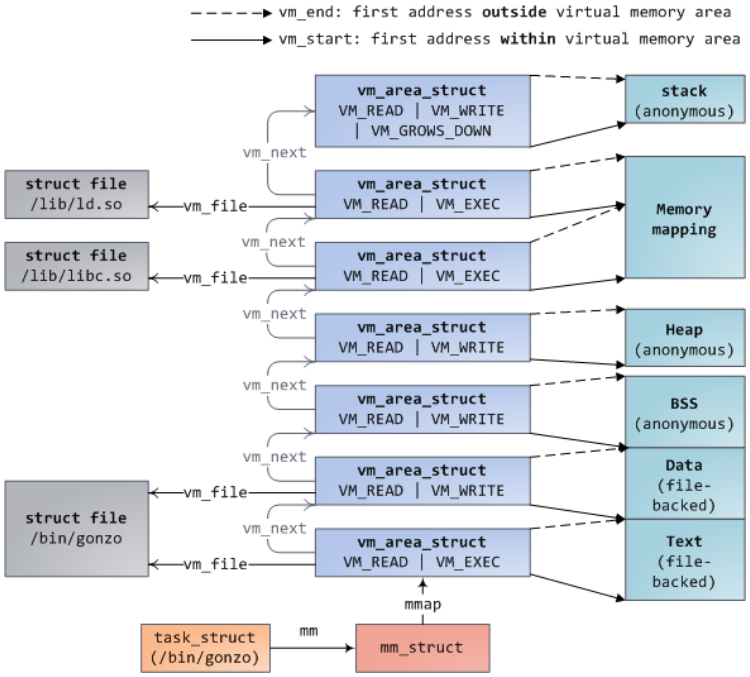

FIGURE *2.5* *Structure des espaces virtuels de mémoire, par G. Duarte*
Source: *`http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/`*

2.2 Buffer overflow

Le dépassement de tampon, « buffer overflow » en anglais, consiste à exploiter une fonction qui ne vérifie par la taille du contenu à copier en mémoire. En utilisant, par exemple, `strcpy()`, on peut redéfinir l'adresse de retour de la fonction et ainsi modifier le flot de contrôle de l'application en le redirigeant à un endroit où l'attaquant aura, par exemple, préalablement injecté du code (« shellcode »).

2.2.1 Stack frame

Une « stack frame » permet de stocker toutes les informations nécessaires à l'exécution d'une fonction. Elle est créée sur la pile lors de l'appel de ladite fonction, par le prologue, et est « détruite », avant l'appel de retour, par l'épilogue. Elle n'est pas détruite à proprement parlé, le pointeur `%ebp` est restauré dans son état précédent.

Lorsqu'une « stack frame » est créée, celle-ci stocke dans un schéma particulier les informations dont elle a besoin. Les premières informations ont des adresses plus grandes en mémoires que les dernières, car la pile grandit de manière décroissante. La séquence suivante est exécutée à chaque prologue de fonction afin d'initialiser la « frame » ; sont placés sur la pile successivement :

1. les paramètres passés à la fonction
2. l'adresse de retour, équivalant à l'adresse de la fonction appelante
3. une sauvegarde du pointeur `%ebp`, pour restaurer la « frame » précédente lors de l'épilogue
4. puis les variables locales, déclarées au sein de la fonction

Le code d'exemple Listing 2.1 contient une fonction `func()` qui est appelée avec deux arguments, `512` et `65536`, et qui déclare des variables locales. La Figure 2.6 illustre l'état de la pile à la fin de la ligne 6, avant le retour de la fonction.

```
1  #include <stdlib.h>
2
3  void func(int param1, int param2)
4  {
5    int local1 = 8;
6    char local_buffer[8] = "foobar";
7  }
8
9  int main(int argc, char **argv)
10 {
11   func(512, 65536);
12   return 0;
13 }
```

Listing 2.1 *Exemple de programme illustrant la gestion de la pile d'exécution*

Les deux arguments de 4 octets (en orange) sont d'abord mis sur la pile, l'adresse de retour ainsi que l'ancienne valeur de `%ebp` de 4 octets [adressage en 32 bits] (en bleu clair) sont ensuite sauvées, puis les deux variables locales, 4 octets pour l'entier et 8 octets pour le `local_buffer` (en vert à droite) sont crées.

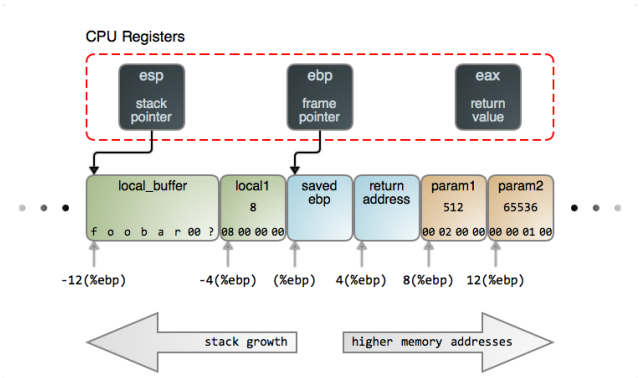

FIGURE *2.6* *Exemple d'une structure de pile (Stack frame), par G. Duarte*
Source: *`http://duartes.org/gustavo/blog/post/journey-to-the-stack/`*

On constate alors que l'adresse de départ du `local_buffer` est inférieur de 16 octets à l'adresse stockant l'adresse de retour. Cette différence est déterministe et ne changera jamais lors de l'exécution.

2.2.2 Exemple

Grâce à cette structure, du fait que la pile grandit avec des adresses décroissantes ainsi que l'utilisation de fonction tel que `strcpy()`, il est possible, en dépassant la taille des variables locales, de modifier des zones mémoires telles que l'adresse de retour. Le code montré en Listing 2.2 permet d'illustrer un cas d'exploitation de la fonction `gets()` (fonction vulnérable car elle ne s'arrête que lorsqu'elle rencontre un retour à la ligne ou une End Of File (EOF)).

```
1  #include <stdlib.h>
2
3  void doRead()
4  {
5    char buffer[28];
6    gets(buffer);
7  }
8
9  int main(int argc)
10 {
11   doRead();
12 }
```

Listing 2.2 *Exemple de programme illustrant un dépassement de tampon*

En regardant la Figure 2.7 on constate que si l'on écrit 28+4+4 octets = 36 octets dans le `buffer`, les 4 derniers octets auront remplacé l'adresse de retour.

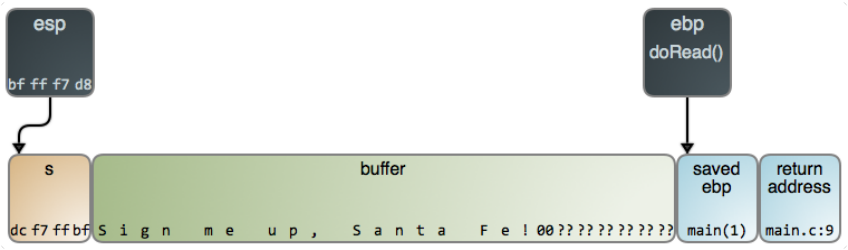

FIGURE 2.7 *Exemple de « stack frame » avant un dépassement de tampon, par G. Duarte*
Source: *http://duartes.org/gustavo/blog/post/epilogues-canaries-buffer-overflows/*

Dans cette exemple, il est possible d'injecter un « shellcode » dans le `buffer` grâce à la fonction `gets()`.

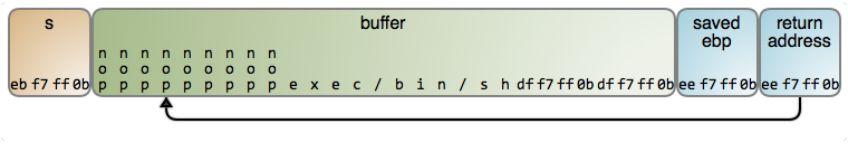

FIGURE 2.8 *Illustration de l'état de la pile au moment d'un dépassement de tampon, par G. Duarte*
Source: *http://duartes.org/gustavo/blog/post/epilogues-canaries-buffer-overflows/*

2.3 DEP/NX

DEP/NX a été mis en place pour éviter, lors d'un dépassement de tampon, que l'attaquant puisse exécuter du code sur la pile. Les endroits mémoire censés contenir des données sont, via les `vm_area_struct` sous Linux, marquées comme étant non-exécutable. Le marquage indique ensuite au processeur, via le NX bit, qu'il ne doit pas exécuter le contenu de cette plage mémoire.

2.3.1 Mécanisme de protection

Le mécanisme de Data Execution Prevention (DEP) 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 des puces PowerPC d'IBM vers l'architecture x86 d'Intel de 2006 à 2007 [8, 9].

La protection en soit se base sur le « hardware », le NX bit, introduit tout d'abord par AMD en 2003, puis repris par Intel sous le nom de XD bit une année après [10, 11]. Ce bit indique au processeur s'il s'agit d'une zone d'instructions ou de données. Cette fonctionalité « hardware » peut aussi être simulée, mais cela entraîne de ce fait une baisse de performance importante.

2.3.2 Contournements grâce aux attaques « return-to-libc »

Une pile non-exécutable ne permet plus à l'attaquant d'exécuter son code, mais cela ne l'empêche pas d'exécuter du code marqué comme exécutable déjà présent au sein du programme ou des librairies dynamiquements chargées. Comme montré dans l'exemple de la Figure 2.5, la bibliothèque partagée **libc** est chargée en mémoire, ce qui est toujours le cas et ce qui rend une attaque de type **« return-to-libc »** [12] possible.

Grâce à la fonction `system()` présente au sein de **libc**, il est possible d'exécuter arbitrairement un programme. Lors de l'attaque on localise, par exemple, une chaîne de caractères tel que `"/bin/sh"`, que l'on prépare comme étant le paramètre à passer à la fonction `system()`.

2.4 ASLR

Comme montré sur la Figure 2.3, l'espace d'adressage virtuel est structuré de manière fixe. Les emplacements mémoires sont donc inchangés à chaque exécution du programme. De cette manière il est possible de prévoir où se trouve en mémoire les différents composants dont a besoin l'attaque. Une attaque de type **« return-to-libc »** a besoin de connaître l'adresse de la fonction `system()` et de la chaîne de caractères `"/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, le mécanisme d'Address Space Layout Randomization est supporté dans le noyau Linux avec la version 2.6.12 [13, 14]. Afin de rendre imprédictible les adresses sensibles, trois décalages aléatoires sont effectués au sein de la mémoire virtuelle. Le premier permet de décaler la pile vers le bas, le second décale lui aussi vers le bas le « Memory Mapping Segment » et le dernier décale vers le haut le segment du tas.

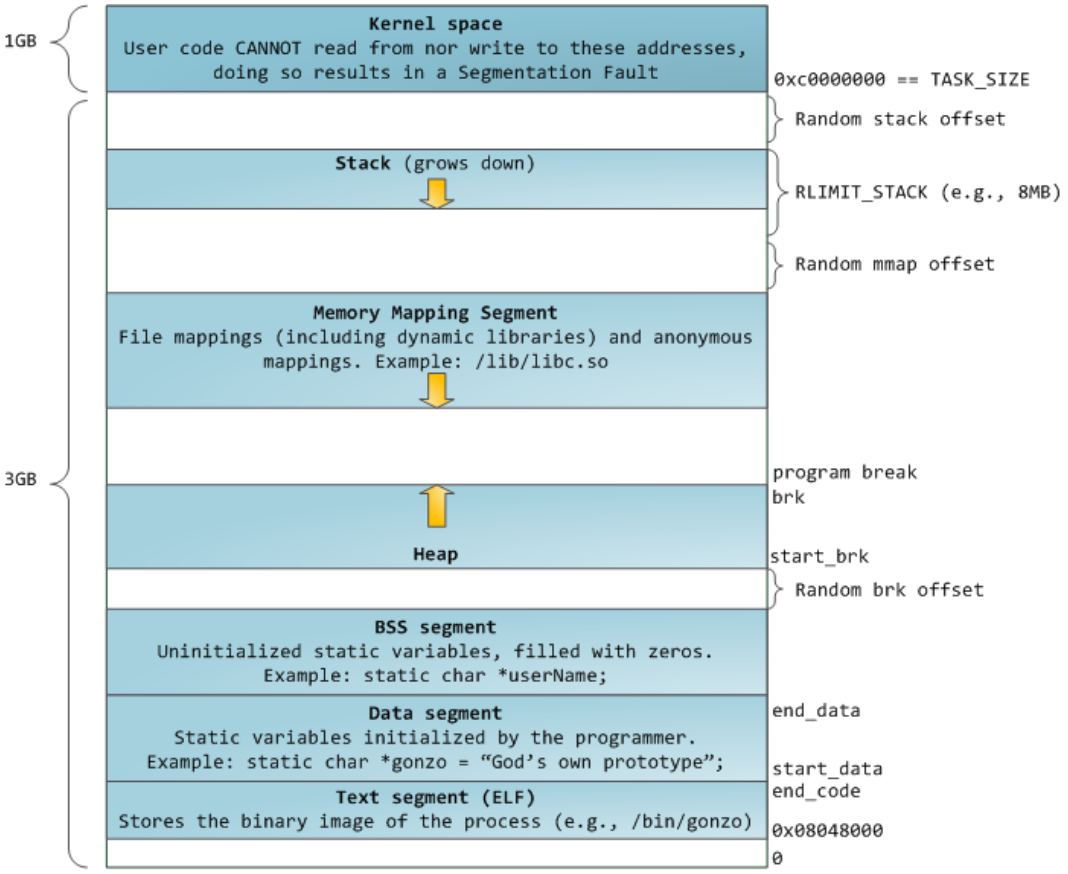

FIGURE *2.9* *Concept de l'address space layout randomization sous Linux en 32 bits, par G. Duarte* ***Source:*** *http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/*

La Figure 2.9 montre bien qu'en 32 bits, l'espace disponible n'est au total que de 4 Go, la part d'aléatoire est donc restrainte. À contrario, dans le cas d'un OS 64 bits ASLR devient bien plus intéressant, car l'espace mémoire virtuel est beaucoup plus vaste (256 To) sans que l'utilisation de celle-ci ne grandisse proportionnellement (au maximum 256 Go de mémoire sont affectés dans des cas classiques d'utilisation serveur). Il donc possible de décaler les segments de manière significative.

Malgré cela, les chercheurs Hector Marco-Gisbert et Ismael Ripoll de l'université de Valence ont écrit un papier démontrant une faiblesse d'ASLR en 64 bits sous certaines hypothèses [15].

2.4.2 Limitation et contournements

Sur un OS 32 bits, la marge de manoeuvre laissée au décalage n'est pas très grande. Seule une partie des bits de l'adresse mémoire est utilisée, ce qui laisse possible une attaque de recherche exaustive réussir en quelques milliers d'essais seulement. En effet la pile est placée aléatoirement avec une entropie de 19 bits seulement et le segment de « Memory Mapping » avec 8 bits.

L'exemple Listing 2.3 montre comment avec un code Python d'une trentaine de lignes il est possible de faire une recherche exhaustive en 32 bits et d'exécuter un « shellcode » dans un programme n'utilisant pas DEP/NX et les « Stack cookies ».

```python
#!/usr/bin/python

import struct, sys, time
from subprocess import PIPE, Popen

# exec /bin/sh
shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\
    \x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"

bufsize = 100
offset = 12     #incl. saved ebp
nopsize = 4096

def prep_buffer(addr_buffer):
    buf = "A" * (bufsize+offset)
    buf += struct.pack("<I",(addr_buffer+bufsize+offset+4))
    buf += "\x90" * nopsize
    buf += shellcode
    return buf

def brute_aslr(buf):
    p = Popen(["./bof", buf]).wait()

if __name__ == "__main__":
    addr_buffer = 0xbf92b39c    # randomly decided
    buf = prep_buffer(addr_buffer)
    i = 0
    while True:
        print i
        brute_aslr(buf)
        i += 1
```

Listing 2.3 Exemple de recherche exhaustive en Python sur ASRL en 32 bits

Ce code est tiré du blog « Sirius CTF » [16] et illustre l'utilisation de l'opération `"\x90"` indiquant au processeur de passer à l'instruction suivante. En définissant une taille de 4096 octets de « No Operation (NOP) » on augmente drastiquement les chances de tomber sur le « shellcode ». La Figure 2.8 montre également l'état de la pile lors d'un dépassement de tampon avec utilisation de l'instruction NOP.

Le but premier d'ASLR est de venir contrer la prédictivité de l'emplacement des segments. On constate cependant que sur la Figure 2.9, aucun décalage n'est appliqué au segments « BSS », « Data » et « Text », ce qui est tout à fait normal, le segment « Text » ne peut pas être placé aléatoirement. Malheureusement cela laisse la prédictibilité de l'emplacement du code exécutable et ouvre la porte à de nouvelles attaques, par exemple de type « Return Oriented Programming (ROP) ».

2.5 Stack canaries

Les « Stack canaries » ou « Stack cookies » sont des valeurs déposées sur la pile d'exécution après la valeur de retour lors de l'appel d'une fonction. Le nom « canaries » vient par analogie des canaris que l'on plaçait dans le mine pour prévenir les fuites de monoxyde de carbone. Ces oiseaux étant petits et vite atteints par les effets du gaz ils donnaient rapidement l'informations aux mineurs qu'un danger était présent [17, 18].

Le fonctionnement des « Stack canaries » est pareil, la valeur du canari est vérifiée lors de l'épilogue de la function et si celle-ci ne correspond pas à la valeur du canari d'origine, alors la tentative de dérouter le flot de contrôle de l'application est détectée, on peut alors réagir en conséquence [19].

FIGURE *2.10* *Illustration de l'état de la pile protégée par un canari, par G. Duarte*
Source: *http://duartes.org/gustavo/blog/post/epilogues-canaries-buffer-overflows/*

2.5.1 Implémentation

Il existe trois types principaux de canaris : « *terminator* », « *random* », et « *random XOR* » [20]. Les « *terminator canaries* » se base sur le constat que l'exploitation d'un dépassement de tampon est une opération sur un chaîne de caractères. De ce fait, si le canari est constitué de caractères tels que `null`, `CR`, `LF` ou encore `-1`, alors la fonction `strcpy()` ou `gets()` se terminera avant de réécrire l'adresse de retour. Le désavantage notable de cette méthode se retrouve dans le fait que l'attaquant connaît la valeur du canari.

Le « *random canary* » est tiré aléatoirement afin de palier au problème du « *terminator canary* ». Généralement ce canari est généré à l'initialisation du programme et est stocké dans une variable globale.

Le « *random XOR canary* » est une méthode un peu plus élaborée, elle fonctionne de la même manière que le « *random canary* » mais est en plus calculée en fonction de tout ou partie du programme, ce qui rend encore plus difficile pour l'attaquant de forger un canari valide.

L'application de la protection par canaries au sein de Clang/LLVM peut se faire grâce au composant « StackProtector » [21]. Ce composant effectue une analyse lors de la compilation, couplée à l'utilisation d'une librairie lors de l'exécution.

2.5.2 Limitation et contournements

Dans les deux premières implémentations (« *terminator canaries* » et « *random canaries* »), il est possible de contourner les canaris en récupérant leur valeur. Ce qui peut être fait directement sur la pile d'exécution grâce au contrôle, par exemple, des paramètres d'une fonction tel que `printf()` qui nous permet de lire en mémoire grâce au format `%p`.

Dans le cas d'un « *random XOR canary* », si l'adresse de retour est modifiée, alors la valeur du canari change, car elle dépend de la valeur aléatoire de départ et des données sur la pile d'exécution. L'exploitation devient alors plus complexe à mettre en place mais reste possible.

2.6 Control-Flow integrity

CFI est un concept permettant, par analyse statique, de définir les changements possibles du flot de contrôle de l'application, appelé « Control-Flow graph (CFG) », pour ensuite valider ou non le changement effectif lors de l'exécution du programme. Le papier original démontrant l'approche [22], publié en 2005, est généralement attribué au centre de recherche de Microsoft. Depuis, de nombreux papiers ont été publiés améliorant les performances ou modifiant légèrement l'approche.

Actuellement le concept de CFI est implémenté au sein de Clang sous le nom de « Forward-Edge CFI » [23, 24, 25]. Dans la documentation dédiée, un example de coût en performance de 15% est relevé sur Chromium. On peut également constater, dans d'autre sources de la litérature, que l'approche classique impose un coût de 16% en moyenne et de 45% dans le pire des cas [26]. Ce qui en fait la principale faiblesse de cette approche.

Les principales variations du concept se font sur la granularité choisie lors de la construction du CFG. Deux « modèles » émèrgent et se généralisent : « Finest-grained CFI » et « Coarse-grained CFI ».

2.6.1 Fine-grained CFI

« Finest-grained CFI » [27, 28], comme son nom l'indique, a une approche précise lors de la construction du CFG. Chaque élément du graphe est labélisé de manière précise et lors de l'exécution du programme, à chaque changement du flot de contrôle on vérifie si la destination labélisée correspond à un label voisin de l'élément de départ. Cependant cette manière de faire amène, par ces nombreuses vérifications, un coût important en terme de performances.

2.6.2 Coarse-grained CFI

« Coarse-grained CFI » est une approche plus simple quant à la labélisation du CFG, et donc plus performante, mais ouvrant la porte à certaines faiblesses. Les points de contrôle de l'application son labélisés de manière à obtenir un nombre restreint de labels. Ceux-ci sont alors réutilisés au sein du même graphe, de ce fait il est possible de partir d'un point A et aller au point C alors que cela n'est pas prévu, mais par cause d'une labélisation non unique l'erreur n'est pas détectée.

2.7 Résumé

Il existe à ce jour une multitude de concepts ayant pour mission d'entravé ou de rallentir les attaques visant le flot de contrôle d'une application. Ces concepts ne gèrent, individuellement, qu'une partie de la problématique et sont donc généralement appliqués conjointement. Aujourd'hui, il n'existe pas encore de mécanismes permettant de garantir seul, à 100%, l'intégrité du flot sans engager des coûts trop importants en matière de performance, rendant leur utilisation impossible dans un environnement de production.

Tous les mécanismes de protection passés en revue peuvent être contourné par une attaque évoluée de type ROP, voir moins évoluée. Le projet Levee promet un mécanisme permettant de contrer ce type d'attaque et à moindre coût. Afin d'y parvenir, l'équipe de chercheurs à mis en place différents concepts se raprochant de ce qui se fait au sein des languages de programmation de type « memory-sage ».

3 Analyse de Levee

Levee est un projet mené dans le cadre du Dependable Systems Lab [29] par les chercheurs Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar et Dawn Song. Le but annoncé du projet est de sécuriser tout programme informatique contre la totalité des attaques de type « control-flow hijack » via une erreur mémoire.

Comme montré dans le chapitre précédent, il existe déjà plusieurs mécanismes (DEP, ASLR) permettant de réduire le risque de ce type d'attaque sans imposer au programme de coût supplémentaire en matière de performance. Cependant, il est aujourd'hui possible de les contourner, même au sein d'un environnement de production, grâce à des attaques évoluées telles que le ROP. D'autres mécanismes plus évolués (CFI) permettent quant à eux d'améliorer fortement la sécurité, mais ne permettent toujours pas de garantir l'entièreté de l'intégrité et reste à ce jour contournable (ROP). Ils ne sont pas adoptés à grande échelle en partie à cause de leurs effets très négatifs sur la performance du programme.

Le défi relevé par l'équipe de recherche a été de proposer un modèle de sécurité entrainant des coûts faibles en matière de performance tout en garantissant l'intégrité complète du flot de contrôle. Levee est le prototype d'implémentation de ce modèle et ce chapitre se concentre sur ses aspects théoriques, son implémentation au sein des outils de compilation LLVM et son rayon d'action.

Contenu du chapitre

3.1 Concepts théoriques

Seules les attaques visant à dérouter le flot de contrôle de l'application sont prises en compte dans le modèle de sécurité. Les attaques du types « data-only », visant à modifier ou récupérer des informations qui ne font pas partie du flot, n'entrent pas en considération.

Ils émettent l'hypothèse 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, les mécanismes de protection résultant de la compilation peuvent se mettre en place avant l'intervention de l'attaquant.

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 flot de contrôle via exploitation d'erreurs mémoires. Dans le cas des languages de types « 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 flot de contrôle impossible, mais entraine une baisse de performances importante.

Afin de garder de bonnes performances tout en garantissant l'intégrité complète, le moins possible d'instrumentation doit être délégué à l'exécution. Le code est alors en premier lieu analysé de manière statique à la compilation, puis des mécanismes de vérification sont ensuite délégués à l'exécution. Le concept de Code-pointer integrity (CPI) [30] intervient alors, afin de determiner quels pointeurs doivent être protégés, ce qui permet de réduire au minimum les accès à controller lors de l'exécution.

3.2 CPI (Code-pointer integrity)

3.2.1 Définition d'un code-pointer

Afin de formaliser les pointeurs de code (Code-pointers), le papier propose la définition suivante :

On dit que l'indirection ou le déréférencement d'un pointeur est sûr si et seulement si l'adresse mémoire à laquelle il tente d'accéder est comprise au sein de l'*objet mémoire* sur lequel le pointeur est *basé*. Un *objet mémoire* est une abstraction liée au language definissant tout types de structures stockées en mémoire (variables globales, variables locales, structures, objets), de sous-structures (un champ d'une structure) ou encore des points de contrôle du flot (adresse de départ d'une fonction, adresse de retour). Ces *objets mémoires* ont un cycle de vie défini, lorsqu'on libére la mémoire liée à un tableau et qu'on alloue un nouveau tableau avec la même adresse, un nouvel *objet mémoire* est créé.

Le papier formalise ensuite la définition de pointeur *basé sur* un *objet mémoire*. On dit qu'un pointeur est *basé sur* sur un *objet mémoire* X si et seulement si le pointeur est obtenu lors de l'exécution par (i) allocation de X sur la pile, (ii) référencement explicit de l'adresse de X, (iii) en référencant l'adresse d'un sous-object y de X (accès au champ y d'une structure X), ou (iv) en effectuant une expréssion sur un pointeur (calcules arithmétiques, indexe de tableau, copie de pointeur) impliquant l'adresse de l'*objet mémoire* X. Cette définition est basée sur la définition des pointeurs *basé sur* du C99.

Si l'on part maintenant du principe que ses deux définitions sont strictement respectées pour tout pointeurs, peu importe les paramètres donnés à l'exécution du programme, alors on peut qualifier le-dit programme de *memory-safe*. Dans ce cas il n'est plus possible de dérouter le flot de contrôle en exploitant une erreur basée sur la mémoire.

3.2.2 Concept théorique

Le mécanisme de CPI est statisfait si toutes les indirections ou les déréférencements sur les pointeurs accédant ou déréférençant des pointeurs jugés « sensibles » sont sûrs. La définition des pointeurs jugés sensibles est donc récursive. Les pointeurs de départ devant être protégés sont les pointeurs responsables de transferer le flot de contrôle de l'application. Cette notion de récursivité est donc dynamique, un pointeur de type `void*` doit être considéré comme sensible s'il pointe vers une fonction ou sur un autre pointeur jugé sensible, mais ne doit pas être considéré comme sensible s'il pointe vers un type `int`.

Les languages de programmation bénéfissiant d'une mémoire supervisée satisfont le concept de CPI. Cependant, ils ne différencient pas les pointeurs sensibles des autres, ce qui entraine une forte baisse de performance par rapport aux languages de type C/C++ (le papier mentionne une perte de $\geq 2\times$ au sein des meilleures implémentations actuelles).

L'observation faite par l'équipe de chercheurs est la suivante : afin de garantir le flot de contrôle de l'application, seul l'accès aux pointeurs jugés sensibles doit être vérifié et ces pointeurs forment un ensemble de petite taille sur l'ensemble des pointeurs. Les pointeurs agissant sur les données peuvent se permettre de ne pas être controllés, ce qui augmente l'efficacité tout en maintenant un très haut taux de protection.

3.2.3 Déterminer l'ensemble des pointeurs sensibles

Afin de déterminer l'ensemble des pointeurs devant être considérés comme sensibles, une analyse statique est effectuée. Pour ce faire une heuristique simple est utilisée : « un pointeur est jugé sensible si son type est sensible ». Les types marqués comme étant sensibles sont :

1. pointeurs de fonction
2. pointeurs vers des pointeurs sensibles
3. pointeurs vers des types composés (`array`, `struct`) qui contiennent un ou plusieurs pointeurs sensibles
4. pointeurs universels (`void*`, `char*`)
5. pointeurs explicitements créés par le compilateur ou à l'exécution (adresse de retour, tables virtuelles en C++ [31], `setjmp` [32])

Cet ensemble de types peut être adapté, étendu, suivant les besoins du programme ou de la plateforme. Le papier mentionne en exemple la structure `ucred` utilisée dans le noyau FreeBSD, qui est marquée comme sensible car elle contient les UIDs des processus et d'autres informations.

En effectuant des tests, les chercheurs se sont rendu compte que l'analyse heuristique mettait en avant un ensemble de pointeurs correspondant en moyenne à 6.5% de l'ensemble global. Ce chiffre met en avant l'impact sur l'efficacité qu'a cette approche.

Le coût en performance dépend directement de l'heuristique et de sa récursion. L'équipe mentione elle-même que toutes heuristiques donnant un ensemble approximé cohérent peuvent être utilisées. Des améliorations peuvent encore être attendues sur ce point. Le second avantage majeur de cette approche est que l'on peut facillement imaginer une nouvelle heuristique permettant de protéger un autre ensemble de pointeurs, pour par exemple protéger une partie des données au sein du programme.

3.2.4 Zone de mémoire sûre « safe region »

Afin de garantir l'intégrité de l'ensemble des pointeurs sensibles, une zone mémoire appelée « *safe region* » est créée, voir la Figure 3.1. Cette région est isolée et ne peut être accédée que via des instructions fournies par CPI. Cela permet de garantir l'intégrité et la propagation des métadonnées. Elle est ensuite constituée d'un « *safe pointer store* », pour les pointeurs sensibles, et de « *safe stacks* », pour une gestion plus spécifique liée à la pile d'exécution, voir la section 3.4. Seule une des deux copies est utilisée suivant le type du pointeur en question.

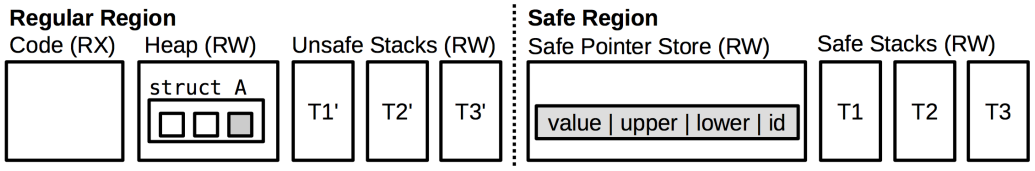

FIGURE *3.1* *Agencement de la mémoire avec CPI, tiré du papier Code-Pointer Integrity*
Source: *`http://dslab.epfl.ch/pubs/cpi.pdf`*

Le *safe pointer store* permet de stocker la valeur du pointeur et les métadonnées de l'*objet mémoire* sur lequel le pointeur est *basé*. Les métadonnées sont constituées de l'adresse de départ et de fin de l'objet au sein de la mémoire standard ainsi que d'un identifiant temporaire de l'objet en question. L'approche est similaire à celle utilisée au sein de SoftBounds+CETS [33] à la différence prêt que CPI stocke également la valeur du pointeur.

CPI garantit (i) que tous les pointeurs sensibles sont stockés dans la « *safe region* », (ii) que la création et la modification lors de l'exécution de pointeurs sensibles soit propagée et (iii) que toutes les indirections ou les déréférences soient vérifiées grâce au métadonnées. Afin d'y parvenir, CPI, lors de la phase de compilation, réécrit les instructions de création et d'accès de l'ensemble des pointeurs sensibles.

3.2.5 Isolation de la zone de mémoire sûre

Une fois les metadonnées et les « *safe stacks* » placées au sein de la région sûre, il faut pouvoir garantir leur intégrité. Pour ce faire cette région doit être isolée du reste de la mémoire et il ne doit pas être possible de la modifier sans passer par les instructions fournies par CPI. Ce mécanisme de protection est directement dépendant de l'architecture pour laquelle le binaire est compilé.

Architecture 32 bits

Dans le cas d'une architecture x86-32, la « *safe region* » est isolée grâce à l'utilisation des segments mémoires. Un segment mémoire n'étant pas utilisé au sein du programme est utilisé par CPI afin de stocker l'adresse de base de la « *safe region* ». Les autres segments sont configurés afin de rendre l'accès à la région sûre impossible.

La preuve de sécurité sous cette architecture repose sur le fait qu'il n'est pas possible, lors de l'exécution, d'atteindre ou de modifier le segment qui contient la région de mémoire sûre.

Architecture 64 bits

Au sein d'une architecture x86-64 la limitation des segments n'est plus, mais l'architecture met toujours à disposition deux registres de segments mémoire. CPI utilise l'un des deux segments pour stocker l'adresse mémoire de base pour la « *safe region* ». Cette adresse de base est tirée aléatoirement lors du lancement de l'application, ce qui ressemble fortement à l'approche d'ASLR expliquée en section 2.4. Cependant, la différence mentionée dans le papier est qu'il n'existe aucun pointeur vers la « *safe region* » au sein de la région standard, ce qui fait qu'il n'est pas possible de remonter jusqu'à l'adresse de base. Les 48 bits d'adressage offrent donc une sécurité suffisante et rende impraticable une attaque de type « recherche exhaustive ».

La preuve de sécurité, sous architecture x86-64, est donc basée sur des informations cachées. Ils mentionnent le fait que leur model est « leak-proof », à la différence d'ASLR, grâce à l'absence de pointeur vers la « *safe region* » lors de l'execution.

3.3 CPS (Code-pointer separation)

Les chercheurs se sont rendus compte que dans le cas du C++, l'ensemble des pointeurs sensibles peut rapidement devenir trop important pour cause, par exemple, des fonctions virtuelles. Tous les pointeurs vers un objet contenant une fonction viruelle deviennent alors sensibles, ce qui peut amené un coût de performance trop élevé. Pour de palier à ce problème, le concept de Code-pointer separation (CPS) a été mis en place. L'idée est de modifier quelque peu l'approche et de baisser les coûts en performance tout en garantissant le flot de contrôle.

Pour y arriver, l'heuristique utilisée pour définir l'ensemble des pointeurs lors de l'analyse statique ainsi que la définition d'un pointeur *basé sur* sont modifiées. Seul les pointeurs pointant directement sur une destination du flot de contrôle sont protégés, laissant la récursion de côté. Contrairement à l'approche CPI, il est possible de ne pas stocker de métadonnées dans la zone de mémoire sûre, en effet, en évitant par exemple les pointeurs sur des objets, chaque pointeur doit alors correspondre exactement à l'adresse de destination, les bornes ne sont plus nécessaires.

Les accès mémoires prennent alors, dans la majeur partie des cas (exepté pour les pointeurs de type universels), la même quantité de ressources qu'un programme sans CPS. À la différence que les pointeurs sensibles sont accédés et stockés dans la région sûre à la place de leur emplacement original.

Les chiffres avancés au niveau des gains en performance de CPS sont de l'ordre de 4.3× plus rapide que CPI. Le coût mesuré passe alors de 8.4% à 1.9%. Évidemment ces chiffres dépendent de l'ensemble des pointeurs à protéger et donc dirrectement de l'architecture dudit programme.

3.4 Safe Stack

3.5 Implémentation au sein de LLVM

3.5.1 Structure de LLVM

3.5.2 Architecture de Levee

3.6 Rayon d'action

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 du chapitre

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.

fdsf

Références

[1] BugTraq, r00t et Underground.Org. *Smashing The Stack For Fun And Profit*. Rapp. tech. Nov. 1996. URL : http://www-inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf.

[2] daemon9. *Smashing The Stack For Fun And Profit*. URL : http://phrack.org/issues/49/14.html.

[3] 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).

[4] 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).

[5] Gustavo Duarte. *Journey to the Stack, Part I*. URL : http://duartes.org/gustavo/blog/post/journey-to-the-stack/ (visité le 10/03/2014).

[6] *64-bit computing*. URL : https://en.wikipedia.org/wiki/64-bit_computing (visité le 27/04/2017).

[7] *Virtual address space details*. URL : https://en.wikipedia.org/wiki/X86-64#Virtual_address_space_details (visité le 19/04/2017).

[8] *Data Execution Prevention*. URL : https://fr.wikipedia.org/wiki/Data_Execution_Prevention (visité le 09/06/2015).

[9] *PowerPC*. URL : https://fr.wikipedia.org/wiki/PowerPC (visité le 29/09/2016).

[10] *Executable space protection*. URL : https://en.wikipedia.org/wiki/Executable_space_protection (visité le 08/01/2017).

[11] *NX Bit*. URL : https://fr.wikipedia.org/wiki/NX_Bit (visité le 21/03/2017).

[12] *Return-to-libc attack*. URL : https://fr.wikipedia.org/wiki/Return-to-libc_attack (visité le 07/06/2016).

[13] *Address space layout randomization [FR]*. URL : https://fr.wikipedia.org/wiki/Address_space_layout_randomization (visité le 15/02/2016).

[14] *Address space layout randomization [EN]*. URL : https://en.wikipedia.org/wiki/Address_space_layout_randomization (visité le 28/03/2017).

[15] 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.

[16] Taishi Nojima. *Exploiting Simple Buffer Overflow (2) | Shellcode + ASLR Bruteforcing*. URL : http://taishi8117.github.io/2015/11/11/stack-bof-2/ (visité le 11/10/2015).

[17] *Stack canaries*. URL : https://en.wikipedia.org/wiki/Stack_buffer_overflow#Stack_canaries (visité le 30/03/2017).

[18] *Sentinel species*. URL : https://en.wikipedia.org/wiki/Sentinel_species#Historical_examples (visité le 16/06/2016).

[19] Gustavo DUARTE. *Epilogues, Canaries, and Buffer Overflows*. URL : http://duartes.org/gustavo/blog/post/epilogues-canaries-buffer-overflows/ (visité le 19/03/2014).

[20] *Buffer overflow protection*. URL : https://en.wikipedia.org/wiki/Buffer_overflow_protection#Canaries (visité le 24/01/2017).

[21] *LLVM StackProtector*. URL : http://llvm.org/docs/LangRef.html#llvm-stackprotector-intrinsic.

[22] Martín ABADI et al. *Control-Flow Integrity - Principles, Implementations, and Applications*. Rapp. tech. University of California, Microsoft Research, Princeton University. URL : https://www.microsoft.com/en-us/research/wp-content/uploads/2005/11/ccs05.pdf.

[23] *Control Flow Integrity*. URL : https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[24] Artem DINABURG. *Let's talk about CFI : clang edition*. URL : https://blog.trailofbits.com/2016/10/17/lets-talk-about-cfi-clang-edition/ (visité le 17/10/2016).

[25] *Control Flow Integrity Design Documentation - Clang*. URL : https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html.

[26] Michael HICKS. *Control Flow Integrity - Université du Maryland*. URL : https://fr.coursera.org/learn/software-security/lecture/0gs3K/control-flow-integrity.

[27] Mathias PAYER, Antonio BARRESI et Thomas R. GROSS. *Fine-Grained Control-Flow Integrity through Binary Hardening*. Rapp. tech. Purdue University, ETH Zurich. URL : https://hexhive.github.io/publications/files/15DIMVA.pdf.

[28] Xinyang GE et al. *Fine-Grained Control-Flow Integrity for Kernel Software*. Rapp. tech. The Pennsylvania State University, Purdue University. URL : https://nebelwelt.net/publications/files/16EUROSP.pdf.

[29] *Dependable Systems Lab*. URL : http://dslab.epfl.ch/ (visité le 23/04/2017).

[30] 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.

[31] DEVELOPPEZ.COM. *Les fonctions virtuelles en C++ : Types statiques et types dynamiques*. URL : http://apais.developpez.com/tutoriels/c++/fonctions-virtuelles-en-cpp/?page=page_6.

[32] *setjmp.h*. URL : https://en.wikipedia.org/wiki/Setjmp.h (visité le 30/12/2016).

[33] *SoftBound + CETS : Complete and Compatible Full Memory Safety for C*. URL : https://www.cs.rutgers.edu/~santosh.nagarakatte/softbound/.

Glossaire

ASLR Address Space Layout Randomization, technique permettant de rendre aléatoire la position des ségments mémoires. 1, 10, 11, 15, 19

CFG Control-Flow graph. 13

CFI Control-Flow integrity. 1, 13, 15

Clang Clang est un compilateur pour les langages de programmation C, C++ et Objective-C. Son interface de bas niveau utilise les bibliothèques LLVM pour la compilation.. 12, 13

Coarse-grained CFI Coarse-grained CFI est une implémentation simplifiée du principe de Control-Flow integrity, échangeant sécurité contre plus de performances. 1, 13

CPI Code-pointer integrity. 16–19

CPS Code-pointer separation. 19

DEP Data Execution Prevention, technique permettant de marquer un espace vituel de mémoire non-exécutable. 1, 9, 10, 15

EOF End Of File. 8

EPFL École polytechnique fédérale de Lausanne. 1

Finest-grained CFI Finest-grained CFI est une implémentation plus complète du principe de Control-Flow integrity. Garantissant une bonne sécurité mais ayant un coût élevé en performances. 1, 13

Levee Levee est une implémentation des concepts de protection CPI, CPS et Safe Stack. Actuellement, mai 2017, une partie du projet a été intégré au sein de LLVM sous le nom de Safe Stack. 1, 14, 15

LLVM LLVM, à la base Low Level Virtual Machine et maintenant nom à part entière, est une approche divergente aux compilateur tel que GCC et une collection d'outils de compilation. 1, 12, 15

NOP No Operation. 11

NX NX bit pour No-eXecute bit est une technique utilisée dans les processeurs pour dissocier les zones de mémoire contenant des instructions des zones contenant des données. 1, 9, 10

ROP Return Oriented Programming. 11, 14, 15

Stack canaries Les stack canaries sont un synonyme de stack cookies. 12

Stack cookies Les stack cookies, ou stack canaries, sont des valeurs déposées sur la pile d'exécution après la valeur de retour lors de l'appel d'une fonction et son controlées à l'épilogue de la-dite fonction. 1, 10, 12