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
HEIG-VD

Expert externe Johan Wehrli
Snap Switzerland Sàrl

Lausanne, HES-SO//Master, le 5 juin 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
Johan Wehrli, Snap Switzerland Sàrl, expert principal

Lieu, date : _____

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

Remerciements

Je tiens particulièrement à remercier le Prof. Pascal Junod pour son soutient et ces conseils avisés lors de la rédaction de se travail. Il a fait preuve d'une implication bien connue, toujours au service de ses étudiants et d'une grande amabilité. Il a aussi su faire preuve d'une grande disponibilité et d'une réactivité remarquable, un grand merci à lui. Merci à ma femme, qui me soutient dans mes entreprises et durant mes longues soirées de travail. Un clin d'œil aux collègues de classe, avec qui partager ses idées sert toujours à avancer... enfin presque toujours. Merci à tous.

Résumé

a remplir...

Mots clés : Levee, Code-pointer Integrity, Code-pointer Separation, Safe stack, LLVM, Clang, ROP

Table des matières

1 Introduction

Nos programmes sont le plus souvent écrits avec des langages de bas niveau comme 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, le nombre d'attaques capables de modifier le flot de contrôle au sein des principaux logiciels que nous utilisons a augmenté. Etant donné la dangerosité de ce type d'attaques, 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, les « 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% de temps d'exécution supplémentaire).

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 fait 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 la mise en pratique dans un environement réel. Exploiter dans un environement réel certaines des attaques brièvement décrites par la suite peut occuper le volume d'un rapport au moins égal à celui-ci.

La description du fonctionnement de la mémoire est inspirée d'articles tirés du blog de Gustavo Duarte [3, 4, 5]. 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 à l'architecture 64 bits est donné. Les termes anglais sont mentionnés une première fois lors de leurs traductions, le document utilise 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} octets (16 Eo), mais uniquement les 48 bits les moins significatifs, et donc 2^{48} octets (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.

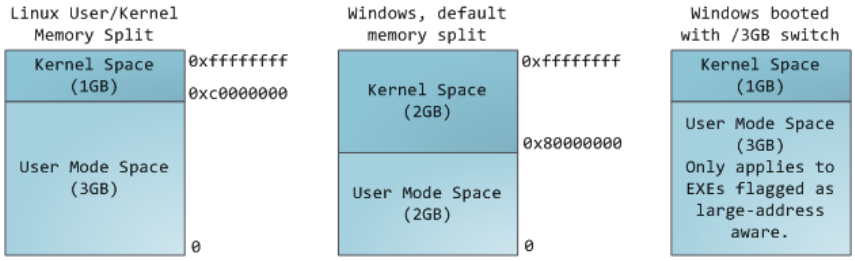

FIGURE 2.1 *Répartition de l'espace mémoire virtuel entre le noyau et le programme, par G. Duarte*
Source: *http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/*

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 :

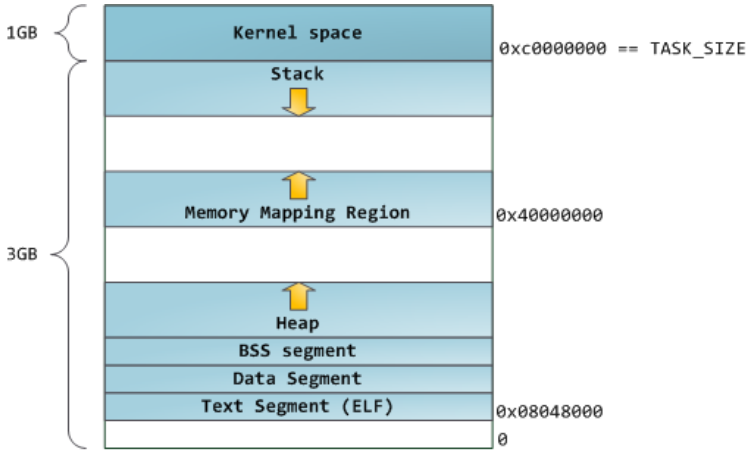

FIGURE 2.2 *Segmentation de la mémoire d'un processus Linux en 32 bits, par G. Duarte*
Source: *http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/*

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 remplit. Il est possible que la pile veuille s'étendre 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 » en anglais, 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 bilbiothèques dynamiques. Il peut aussi être utilisé à d'autres fins, par exemple pour stocker des données, tel le tas.

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. La Figure 2.3 illustre un exemple de ce que l'on peut retrouver dans ces segments.

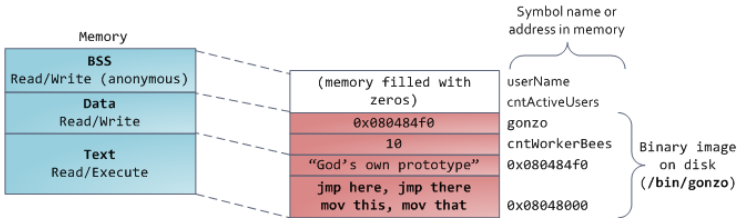

FIGURE 2.3 *Plan d'une image binaire dans les segments BSS, data et text, par G. Duarte*
Source: *http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/*

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.

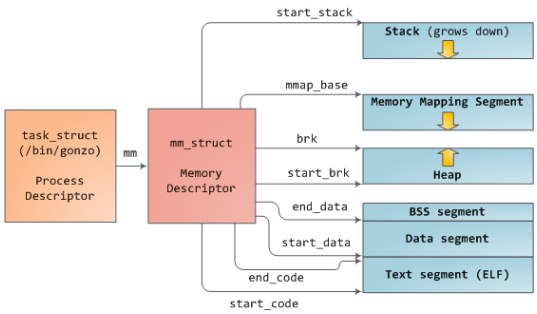

FIGURE 2.4 *Descripteur de mémoire d'un processus Linux, par G. Duarte*
Source: *http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/*

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.

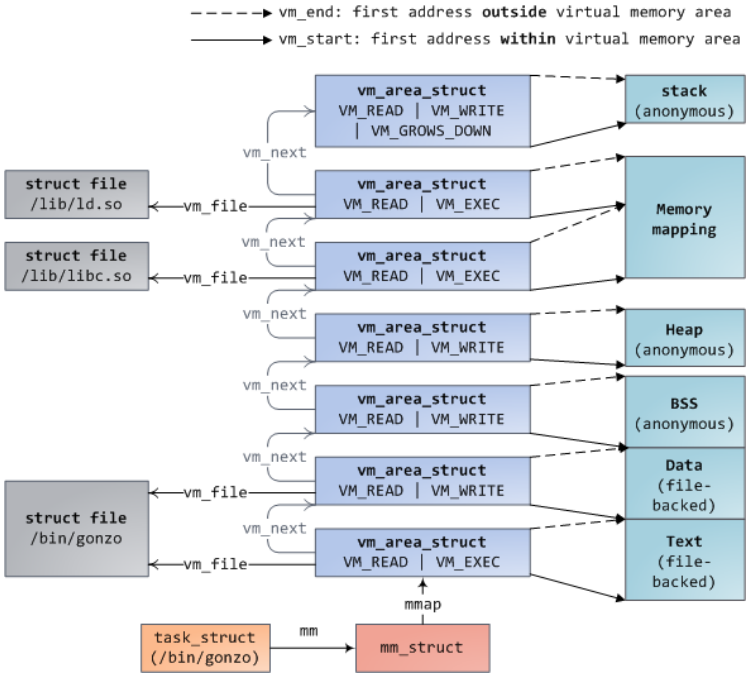

FIGURE *2.5* *Structure des espaces virtuels de mémoire, par G. Duarte*
Source: *`http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/`*

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()`, il est possible d'écrir sur 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.

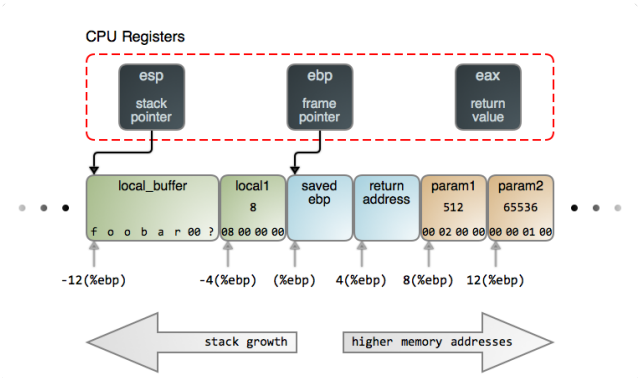

FIGURE *2.6* *Exemple d'une structure de pile (Stack frame), par G. Duarte*
Source: *`http://duartes.org/gustavo/blog/post/journey-to-the-stack/`*

On constate alors que l'adresse de départ du `local_buffer` est inférieure 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 un « 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 écrasé l'adresse de retour.

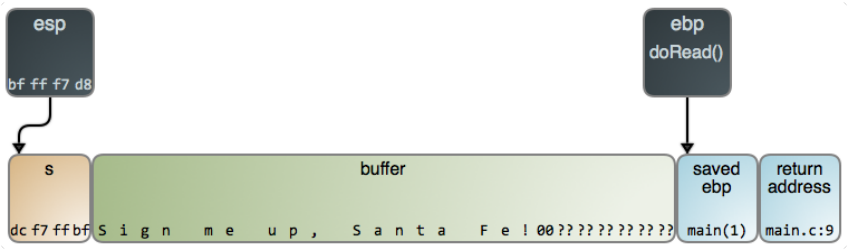

FIGURE 2.7 *Exemple de « stack frame » avant un dépassement de tampon, par G. Duarte*
Source: *http://duartes.org/gustavo/blog/post/epilogues-canaries-buffer-overflows/*

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

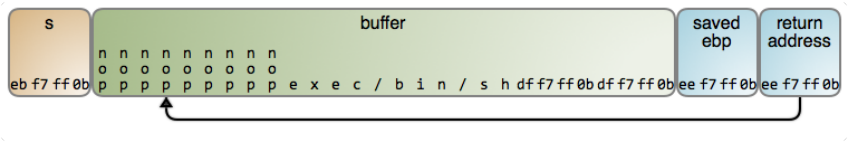

FIGURE 2.8 *Illustration de l'état de la pile au moment d'un dépassement de tampon, par G. Duarte*
Source: *http://duartes.org/gustavo/blog/post/epilogues-canaries-buffer-overflows/*

2.3 DEP/NX

Le mécanisme DEP/NX a été mis en place pour éviter, lors d'un dépassement de tampon, que l'attaquant puisse exécuter du code stocké 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 drapeau NX, 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 bibliothèques 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.

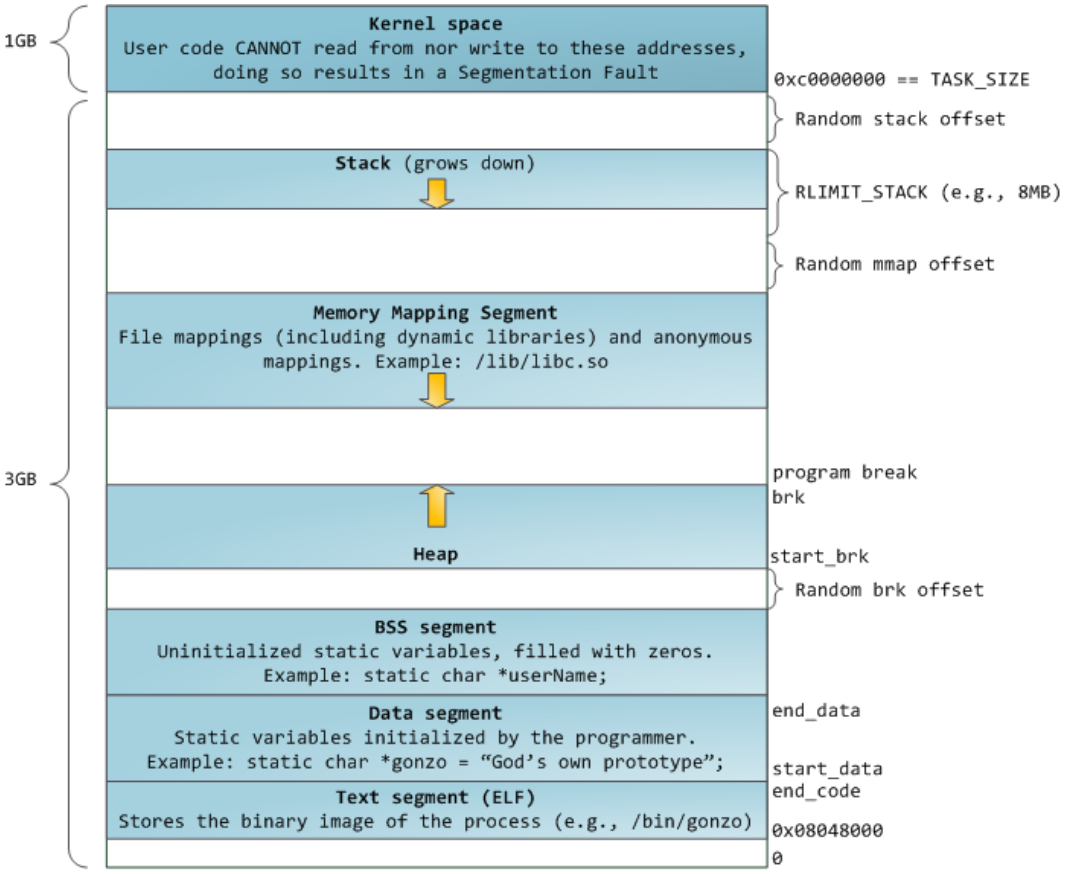

FIGURE *2.9* *Concept de l'address space layout randomization sous Linux en 32 bits, par G. Duarte* ***Source:*** *http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/*

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 restreinte. 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 à une attaque par recherche exaustive, exigeant quelques milliers d'essais seulement, d'aboutir. 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, ASLR ne modifie pas le segment « text ». 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) ». Il existe d'autres mécanismes, tel que « position-independent executable (PIE) » [17], vennant appliquer de l'aléatoire sur certaines partie du code et des bibliothèques. Mais ces mécanismes ne sont pas décrit dans ce rapport.

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 aux canaris que l'on plaçait dans le mine pour prévenir les fuites de monoxyde de carbone [18, 19]. Ces oiseaux étant petits et vite atteints par les effets du gaz, ils donnaient rapidement l'informations aux mineurs qu'un danger était présent.

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 une tentative de dérouter le flot de contrôle de l'application est détectée et l'on peut alors réagir en conséquence [20].

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* » [21]. Les « *terminator canaries* » se base sur le constat que l'exploitation d'un dépassement de tampon est une opération sur une 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 » [22]. 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 [23], 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 » [24, 25, 26]. 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 [27]. 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 »

La variation « finest-grained CFI » [28, 29], 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 »

La variation « 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'entraver ou de ralentir 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 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.

Le papier présentant CPI/CPS/SafeStack [30] et son prototype appelé Levee promettent 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-safe ». En effet, les implémentations actuelles permettant de garantire à 100% le flot de contrôle — SoftBound+CETS, CCured ou encore AddressSanitizer — sont une application directe des concepts présent au sein des languages de type « memory-safe » aux languages `C/C++`. Cependant cette application directe n'est pas viable en terme de performances. Les chercheurs du projet Levee ont alors proposé d'appliquer ces méthodes sur une partie spécifique du programme.

3 Analyse de Levee

Levee est un projet mené dans le cadre du Dependable Systems Lab [31] 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, du moment où celui-ci a été compilé grâce à Clang avec les options nécessaires.

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, SoftBound+CETS, CCured ou encore AddressSanitizer) permettent quant à eux d'améliorer très fortement la sécurité, mais ne permettent pas, soit de garantir une totale intégrité et reste à ce jour contournable, soit ils ne sont pas adoptés à grande échelle à cause de leurs effets beaucoup trop négatifs sur la performance du programme — jusqu'à 116% de coût en performance —.

Le défi relevé par l'équipe de recherche a été de proposer un modèle de sécurité entraînant 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 de contrôle, 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 l'exploitation d'erreurs mémoires. Dans le cas des langages 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 entraîne une baisse de performances importante.

Afin de garder de bonnes performances tout en garantissant l'intégrité complète, un minimum 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 déterminer quels pointeurs doivent être protégés, ce qui permet de réduire au minimum les accès à contrôler lors de l'exécution.

3.2 « Code-pointer integrity » (CPI)

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 langage définissant 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 explicite de l'adresse de X, (iii) en référençant l'adresse d'un sous-objet y de X (accès au champ y d'une structure X), ou (iv) en effectuant une expression sur un pointeur (calculs arithmétiques, position au sein d'un tableau, copie de pointeur) impliquant l'adresse de l'*objet mémoire* X. Cette définition est basée sur la définition des pointeurs *basé sur* présente au sein de la norme `C99`.

Si l'on part maintenant du principe que ces 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 satisfait 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 transférer 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 langages de programmation bénéficiant d'une mémoire supervisée satisfont le concept de CPI. Cependant, ils ne différencient pas les pointeurs sensibles des autres, ce qui entraîne une forte baisse de performance par rapport aux langages 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 contrôlés, 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. les pointeurs de fonction
2. les pointeurs pointant sur des pointeurs sensibles
3. les pointeurs pointant sur des types composés (`array`, `struct`) qui contiennent un ou plusieurs pointeurs sensibles
4. les pointeurs universels (`void*`, `char*`)
5. les pointeurs explicitements créés par le compilateur ou à l'exécution (adresse de retour, tables virtuelles en `C++` [32], `setjmp` [33])

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 mentionne elle-même que toutes les heuristiques donnant un ensemble approximatif cohérent peuvent être utilisées. Des améliorations peuvent encore être attendues sur ce point. Le second avantage majeur de cette approche réside dans le fait que l'on peut facilement imaginer de nouvelles heuristiques permettant de protéger d'autres ensembles de pointeurs, par exemple, pour protéger une partie des données au sein du programme.

3.2.4 Zone de mémoire sûre

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 des données 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 l'une des deux copies présente dans les deux régions est utilisée suivant le type du pointeur en question.

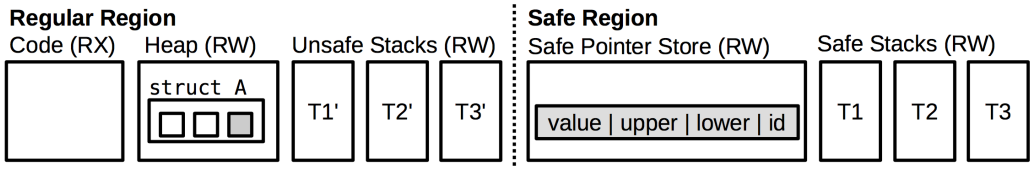

FIGURE *3.1* *Agencement de la mémoire avec CPI, tiré du papier Code-Pointer Integrity*
Source: *http://dslab.epfl.ch/pubs/cpi.pdf*

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 [34] à la différence prêt que CPI stocke également la valeur du pointeur et n'applique pas de protection sur tout les pointeurs du programme.

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 mémoire sûre

Une fois les métadonnées 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. Dans tous les cas, la région sûre est placée dans le segment de « memory mapping » avec un accès en lecture et écriture.

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 présente, 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 rendent impraticable une attaque par recherche exhaustive.

La preuve de sécurité, sous architecture `x86-64`, est donc basée sur des informations cachées. Ils mentionnent le fait que leur modèle est « leak-proof » — c'est-à-dire qu'il ne laisse pas fuiter d'informations concernant la localisation de la région sûre —, à la différence d'ASLR, grâce à l'absence de pointeur vers la « *safe region* » lors de l'exécution.

3.3 « Code-pointer separation » (CPS)

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, d'utilisation de fonctions virtuelles. Tous les pointeurs vers un objet contenant une fonction virtuelle deviennent alors sensibles, ce qui peut induire un coût de performance trop élevé. Pour 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 (excepté pour les pointeurs de type universel), 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 »

Le composant « safe stack » est un équivalant, mais potentiellement plus complet, aux « stack cookies ». Son but est de protéger les variables sensibles stockées sur la pile d'exécution. Pour ce faire, à la manière dont fonctionne CPI, « safe stack » sépare les variables dans deux régions mémoires différentes. Cependant, dans le cas de « safe stack », l'approche est différentes. Seules sont déplacées les variables dont l'allocation ou les accès sont considérés comme non-sécurisés. Les allocations et les accès à ces variables sont ensuite remplacés lors de la phase de compilation par « safe stack » afin de les initialiser dans une zone mémoire non protégée. Cette zone mémoire est initialisée au lancement de l'application, au sein du segment de « memory mapping ».

Comme montré sur la 3.1, la zone mémoire sûre, ou région sûre, est constituée d'une ou plusieurs « safe stacks ». Chaque « safe stack » appartient à un `thread`, celui-ci contient un pointeur vers la pile standard et un pointeur vers la pile sûre. Le registre `%esp` sur l'architecture `x86-32` pointe vers la « safe stack »— `%rsp` sur `x86-64` —, tandis qu'une variable locale au `thread` pointe vers la « unsafe stack », présente dans le segment de « memory mapping ».

3.4.1 Analyse à la compilation

Il est nécessaire de passer par un phase d'analyse afin de determiner, comme pour CPI, quelles variables déplacer. La fonction `IsSafeStackAlloca()` détermine quels types de variables peuvent être accédés de manière non-sécurisée. Une variable est déclarée non-sécurisée si l'un des critères est retenu :

1. un pointeur vers la variable est stocké quelque part en mémoire ;
2. un élément appartenant à un tableau est accédé par un index non-constant (une autre variable) ;
3. un tableau dynamique est accédé (via une variable ou un index constant) ;
4. un pointeur vers la variable est donnée en arguement à une fonction.

3.4.2 Support à l'exécution

Afin d'initialiser et gérer les allocations mémoires de la zone non-sécurisée, une bibliothèque est fournie à l'exécution. Plus de détails sont donnés dans la sous-section 3.5.4.

3.5 Implémentation au sein de LLVM

La première implémentation de Levee a été portée au sein de la version 3.3 de LLVM (juin 2013). Pour ce prototype, l'équipe de Levee a dû modifier quelque peu l'infrastructure, des modifications ont été faites sur les bibliothèques de LLVM ainsi que dans Clang. Depuis la version 3.8, en mars 2016, « safe stack » est disponible au sein de LLVM et Clang, cette intégration a été faite par un employé de Google,

Peter Collingbourne. Actuellement, en mai 2017, dans la version 4.0 de LLVM, les prototypes de CPI et CPS ne sont toujours pas présents et aucune date d'intégration n'a été annoncée.

3.5.1 Historique de LLVM

Le projet débute dans le début des années 2000, par ce qui est au début un travail de recherche universitaire de Chris Lattner [35]. À cette époque, il est courant de voir pour les compilateurs une implémentation monolitique, par exemple GCC pour la famille des langages `C`. Au sein de ces implémentations, Lattner remarque qu'il est également difficile de réutiliser du code pour servir différents intérêts, utiliser l'analyseur — « parser »— de GCC pour effectuer une analyse statique n'est pas possible et les parties de code en commun sont relativement petites.

Lattner propose alors une collection, open source, souple et réutilisable, de technologies permettant de développer des « frontends » et « backends » de compilateur. Le design le plus populaire utilisé par les compilateurs est constitué de trois phases, (i) le « frontend », responsable d'analyser le code et de générer une représentation abstraite — appelée « abstract syntax tree (AST) » — (ii) l'« optimizer », améliore cette représentation (suppression des redondances, simplification des opérations mathématiques, etc.), puis (iii) le « backend », transcrit la représentation intermédiaire en code machine suivant la plateforme ciblée.

FIGURE *3.2* *Composants majeurs d'un compilateur en trois phases*
Source: *http://www.aosabook.org/en/llvm.html*

Il y a plusieurs avantages majeurs à cette approche, premièrement lorsque l'on souhaite supporter un nouveau langage, seul le « frontend » doit être écrit. De la même manière, si l'on souhaite rajouter une plateforme, les « frontend » existant peuvent facilement en profiter. Deuxièmement, au niveau des projets open source, il est très avantageux d'avoir, au sein du même projet, différents langages ciblés, car cela accroît la communauté potentielle et donc le nombre de contributeurs. En finalité, le fait qu'il n'est pas nécessaire d'avoir les mêmes compétences pour écrire un « frontend », un « optimizer » ou un « backend » aide à trouver des contributeurs, un projet open source veut réduire au maximum le pas à franchir pour contribuer.

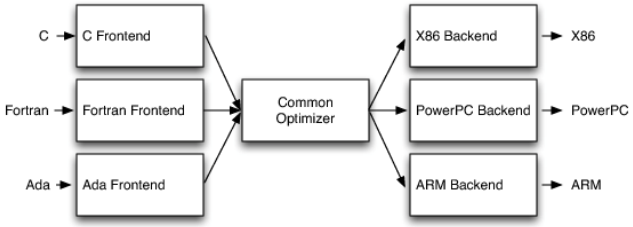

FIGURE *3.3* *Modularité d'une structure en trois phases*
Source: *http://www.aosabook.org/en/llvm.html*

Dans un premier temps, LLVM est l'acronyme de « low level virtual machine », mais rapidement le projet se développe et le nom LLVM devient une marque rassemblant sous son nom différents projets tel que LLVM intermediate representation (LLVM IR), LLDB, LLD ou encore LLVM `C++ Standard Library`, une implémentation de la bibliothèque standard `C++11` et `C++14`.

3.5.2 Structure de LLVM

LLVM reprend le design des trois phases et l'implémente en plusieurs parties. Parmi ces parties, le langage intermédiaire joue un très grand rôle. C'est lui qui donne le plus de sens à l'architecture choisie et qui rend si puissant LLVM.

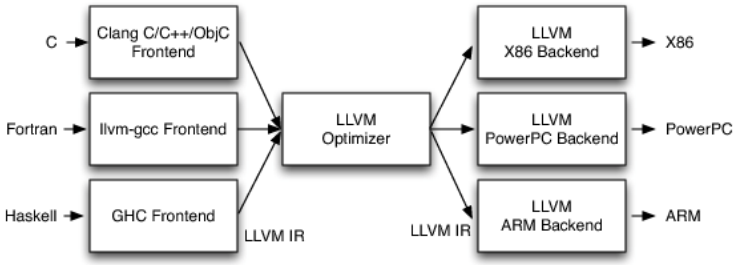

FIGURE *3.4* *Vue d'ensemble des composants de LLVM*
Source: *http://www.aosabook.org/en/llvm.html*

Représentation intermédiaire

LLVM IR est un langage en soit, normé et bien défini, fortement typé, ayant pour seul but de répondre aux besoins de l'optimisation. Sa définition en fait un langage permettant d'être très fortement modifié et restructuré lors de la phase en question.

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}

define i32 @add2(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse

recurse:
  %tmp2 = sub i32 %a, 1
  %tmp3 = add i32 %b, 1
  %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
  ret i32 %tmp4

done:
  ret i32 %b
}
```

Listing 3.1 *Example de code LLVM IR, source : http://www.aosabook.org/en/llvm.html*

La représentation Listing 3.2 est le résultat du code `C` en Listing 3.1. Le défi réside dans la propagation des informations, lorsque l'on transforme du code `C` en une quelconque autre représentation, certaines informations peuvent être perdues.

Lorsque l'équipe de Levee a commencé à travailler sur le projet, ils se sont rendu compte que certaines informations étaient perdues lors de la transformation en LLVM IR et ont dû compléter les métadonnées émises par Clang.

```c
unsigned add1(unsigned a, unsigned b) {
  return a+b;
}

// Perhaps not the most efficient way to add two numbers.
unsigned add2(unsigned a, unsigned b) {
  if (a == 0) return b;
  return add2(a-1, b+1);
}
```

Listing 3.2 *Code source `C` de l'exemple de représentation intermédiaire, source : `http://www.aosabook.org/en/llvm.html`*

Le langage intermédiaire LLVM IR est constitué d'instructions ressemblant fortement au langage machine, mais à quelques différences. De manière non-exhaustive, (i) les conventions d'appel de `ret` ou `call` sont abstraites — c'est au « backend » de se soucier de ce genre de conventions —, (ii) un nombre infini de variables commençant par `%` est à disposition et (iii) celles-ci sont typées grâce à un système simple : `i32` pour un entier de 32 bits, `i32**` pour un pointeur de pointeur sur un entier de 32 bits, et cetera.

LLVM IR peut être stocké et manipulé sous trois forme différentes : (i) la forme textuelle, comme montrée en Listing 3.1, (ii) sous forme de données structurées en mémoire — forme utilisée par les optimiseurs — et (iii) sous une forme de « bitcode », sauvegardée sur le disque, permettant une gestion plus efficace et plus dense de l'information. Des outils de conversion du « bitcode » (`.bc`) au format textuel (`.ll`) et vice versa sont présents dans la suite LLVM.

LLVM « frontends »

LLVM IR est la seule interface au sein de LLVM, ce qui signifie que le seul prérequis pour écrire un « frontend » LLVM est de connaître son fonctionnement. Du fait que ce langage soit un langage textuel, il suffit au « frontend » d'écrire le résultat dans un fichier texte et une simple commande tel que les « Unix pipes » permettent de profiter de tout la suite de l'infrastructure.

LLVM « optimizers »

Le fonctionnement d'un « optimizer » est plutôt simple : il prend en entrée le code sous forme de la représentation intermédiaire et produit du code, toujours sous la forme LLVM IR, normalement plus efficace. Les « optimizers » sont organisés sous forme de pipeline, chaque « optimizer » représente une « pass » dans celui-ci. Suivant le niveau d'optimisation souhaité lors de la compilation, par exemple `-O3`, le plus haut niveau, le nombre de « pass » fluctue.

Chaque « pass » au sein de LLVM est écrite en `C++` et dérive — indirectement — de la classe `Pass`. Les « pass » doivent être les plus indépendantes possible les unes des autres, leur architecture permet d'inclure seulement les « pass » utilisées lors de la compilation.

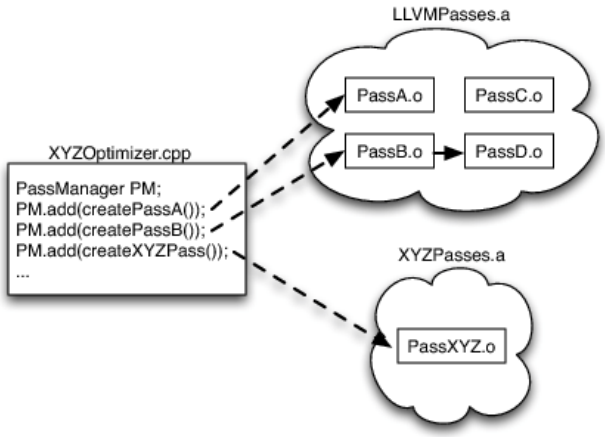

FIGURE *3.5* *Exemple de construction d'un optimiseur LLVM*
Source: *http://www.aosabook.org/en/llvm.html*

L'interface, très claire, des « pass » permet une très grande modularité ainsi qu'une certaine puissance et cela laisse la possibilité d'implémenter n'importe quelle fonctionnalité au sein de LLVM.

LLVM « backends »

La transcription entre le langage intermédiaire et le code machine est géré par le composant « LLVM code generator ». Il est nécessaire de produire le code le plus spécifique et le plus optimisé sur chaque plateforme, de ce fait il est préférable que chaque « backend » produise un code machine unique. Cependant, il est aussi intéressant de pouvoir partager une partie de la logique parmi — par exemple l'allocation des variables % au sein des registres —. Pour ce faire, le système de « pass » est également appliqué lors de la génération du code machine.

LLVM met à disposition une liste de « pass » présentes par défaut, répondant à des problématiques récurentes, ainsi l'auteur de la plateforme ciblée peut choisir de redéfinir certaine « pass », de les enlever ou d'en rajouter.

Cette approche amène alors le défi de fournir des « pass » génériques, mais spécifiques à chaque plateforme. LLVM répond à ce défi grâce à un langage spécifique, les « LLVM target description files » (`.td`), permettant de fournir, pour chaque plateforme, la description de celle-ci pour la « pass » en question.

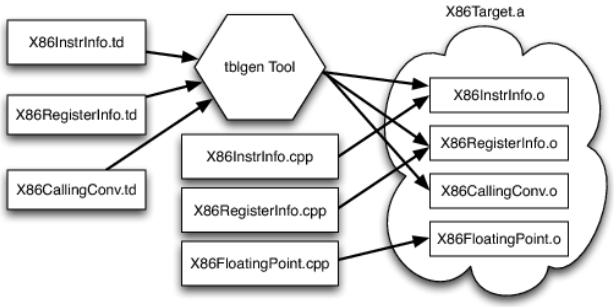

FIGURE *3.6* *Simplification du processus de définition spécifique à la plateforme* ***x86***
Source: *http://www.aosabook.org/en/llvm.html*

```
def GR32 : RegisterClass<[i32], 32,
  [EAX, ECX, EDX, ESI, EDI, EBX, EBP, ESP,
   R8D, R9D, R10D, R11D, R14D, R15D, R12D, R13D]> { ... }
```

Listing 3.3 Description des registres disponibles pour le « backend » `x86`

Le Listing 3.3 est un exemple de `.td` décrivant les registres à disposition pour la plateforme `x86`. Différentes informations sont présentes, par exemple le fait que les registres sont en 32 bits, de préférence alignés, etc. D'autres informations tel que l'ordre d'utilisation préféré sont décrites dans ces fichiers.

3.5.3 Implémentation de CPI

L'implémentation de CPI commme Levee le propose au sein de la version 3.3 de LLVM est composée de deux « pass ». La première permettant d'identifier les pointeurs sensibles selon le design expliqué en sous-section 3.2.3 et la deuxième permettant de mettre en place la séparation des pointeurs et d'assurer leur intégrité commen expliqué en sous-section 3.2.4 et en sous-section 3.2.5.

Comme mentioné précédement, il se peut que de l'information soit perdue lors de la transformation du code source en LLVM IR. Dans le cas de cette implémentation, le cas de figure c'est imposé. Clang ne conservait pas le type original des pointeurs étant transformé en `void*` quand ceux-ci sont passés à la fonction `memset` par exemple. Ou encore que LLVM IR ne faisait pas la différence entre un pointeur de type `void*` et `char*` — les deux étant représenté par un `i8*` —, ce qui est une information utile dans le cas de CPI. L'équipe a donc dû améliorer Clang afin de préserver ces informations lors du passage du `C/C++` au LLVM IR entant que « LLVM metadata ».

La seconde « pass » permettant de mettre en place l'instrumentation rajoute, dans la plupart des cas, des appels aux fonctions tels que `cpi_ptr_store()` ou encore `cpi_memcopy()`. Ces fonctions sont implémentées au sein du « `compiler-rt` » et sont compilées entant que « LLVM bitcode » puis liées aux objets compilés par Clang.

3.5.4 Architecture de « safe stack »

L'implémentation de « safe stack », dans la version 4.0 de LLVM, est découpée en deux parties : (i) une « pass » d'analyse et d'optimisation et (ii) l'implémentation de la bibliothèque d'exécution.

Anaylse et optimisation

Une seule « pass » au sein de l'« optimizer » est résponsable d'appliquer « safe stack » et est présente sous le nom de `SafeStackLegacyPass`.

La classe implémente plusieurs méthodes dont (i) une méthode permettant de vérifier une allocation donnée, (ii) une méthode permettant de déplacer les variables non-sécurisée statiques, (iii) une méthode permettant de déplacer les variables non-sécurisée dynamiques ou encore (iv) une méthode permettant d'appliquer « safe stack » sur une fonction donnée.

```
/// The SafeStack pass splits the stack of each function into the safe
/// stack, which is only accessed through memory safe dereferences (as
/// determined statically), and the unsafe stack, which contains all
/// local variables that are accessed in ways that we can't prove to
/// be safe.
class SafeStackLegacyPass : public FunctionPass {

  /// \brief Allocate space for all static allocas in \p StaticAllocas,
  /// replace allocas with pointers into the unsafe stack and generate code to
  /// restore the stack pointer before all return instructions in \p Returns.
  ///
  /// \returns A pointer to the top of the unsafe stack after all unsafe static
  /// allocas are allocated.
  Value *moveStaticAllocasToUnsafeStack(IRBuilder<> &IRB, Function &F,
                                        ArrayRef<AllocaInst *> StaticAllocas,
                                        ArrayRef<Argument *> ByValArguments,
                                        ArrayRef<ReturnInst *> Returns,
                                        Instruction *BasePointer,
                                        AllocaInst *StackGuardSlot);

  /// \brief Replace all allocas in \p DynamicAllocas with code to allocate
  /// space dynamically on the unsafe stack and store the dynamic unsafe stack
  /// top to \p DynamicTop if non-null.
  void moveDynamicAllocasToUnsafeStack(Function &F, Value *UnsafeStackPtr,
                                       AllocaInst *DynamicTop,
                                       ArrayRef<AllocaInst *> DynamicAllocas);

  bool IsSafeStackAlloca(const Value *AllocaPtr, uint64_t AllocaSize);

  bool IsMemIntrinsicSafe(const MemIntrinsic *MI, const Use &U,
                          const Value *AllocaPtr, uint64_t AllocaSize);

  bool IsAccessSafe(Value *Addr, uint64_t Size, const Value *AllocaPtr,
                    uint64_t AllocaSize);

public:
        bool runOnFunction(Function &F) override;
}
```

Listing 3.4 *Déclaration partielle de la classe « safe stack » montrant certaines des méthodes mentionnées*

La méthode `runOnFunction()` permet de séparer la pile d'exécution, si besoin est, de la fonction passée en paramètre. C'est le point de départ de l'analyse et de l'instrumentation de « safe stack ». Cette méthode fait appel à `IsSafeStackAlloca()`, voir Listing 3.5, permettant de determiner si une allocation doit être déplacée ou non, selon les explications en sous-section 3.4.1.

```
/// Check whether a given allocation must be put on the safe
/// stack or not. The function analyzes all uses of AI and checks whether it is
/// only accessed in a memory safe way (as decided statically).
bool SafeStack::IsSafeStackAlloca(const Value *AllocaPtr, uint64_t AllocaSize);
```

Listing 3.5 *Signature de la méthode permettant de déterminer si une allocation est sûre*

La méthode `moveStaticAllocasToUnsafeStack()`, voir Listing 3.6, ainsi que la méthode `moveDynamicAllocasToUnsafeStack()` permettent de générer et remplacer le code nécessaire afin de modifier les allocations marquées par la méthode `IsSafeStackAlloca()`. Lors de l'appel à la méthode `runOnFunction()` les allocations

des variables statiques sont dabord modifiées, puis un pointeur vers le haut de la pile non-sécurisée est récuperé avant de modifier les allocations dynamiques, partant dudit pointeur.

```
/// \brief Allocate space for all static allocas in \p StaticAllocas,
/// replace allocas with pointers into the unsafe stack and generate code to
/// restore the stack pointer before all return instructions in \p Returns.
///
/// \returns A pointer to the top of the unsafe stack after all unsafe static
/// allocas are allocated.
Value *moveStaticAllocasToUnsafeStack(IRBuilder<> &IRB, Function &F,
                                      ArrayRef<AllocaInst *> StaticAllocas,
                                      ArrayRef<Argument *> ByValArguments,
                                      ArrayRef<ReturnInst *> Returns,
                                      Instruction *BasePointer,
                                      AllocaInst *StackGuardSlot);

/// \brief Replace all allocas in \p DynamicAllocas with code to allocate
/// space dynamically on the unsafe stack and store the dynamic unsafe stack
/// top to \p DynamicTop if non-null.
void moveDynamicAllocasToUnsafeStack(Function &F, Value *UnsafeStackPtr,
                                     AllocaInst *DynamicTop,
                                     ArrayRef<AllocaInst *> DynamicAllocas);
```

Listing 3.6 *Signatures des méthodes permettant de déplacer les allocations dans la zone non-sécurisée*

Afin de lire et comprendre toute l'implémentation de « safe stack » dans la version 4.0, vous trouverez « safe stack » sur le dépôt miroire de LLVM disponible ici. Le commit ajoutant « safe stack » à LLVM est disponible ici.

Bibliothèque d'exécution

Comme pour CPI, « safe stack » est dotée d'une bibliothèque permettant de géréer son initialisation. On retrouve dans cette bilbiothèque les fonctions s'occupant d'intercépter les événements des `threads` afin d'initialiser et libérer les piles d'exécution non-sécurisées, ainsi que les fonctions de gestion de la pile, voir Listing B.3. L'implémentation complète est disponible, dans sa version 4.0, ici.

3.6 Rayon d'action

La combinaison CPI et « safe stack » permet effectivement de couvrire tout les attaques contre le flot de contôle se basant sur une erreur provenant de la gestion de la mémoire. La protection « safe stack » seule, tel qu'implémentée au sein de LLVM actuellement, permet de se prémunir contre les dépassements de tampon ayant lieu uniquement sur la pile d'exécution. « Safe stack » ne couvre donc qu'une partie des attaques connues au jour d'aujourd'hui.

3.6.1 « Safe stack » ou canaris

« Safe stack » est désigné comme équivalant, voir en remplacement des canaris. La question se pose alors, quels sont les avantages de « safe stack » sur les canaris ? La mise en pratique d'une attaque, voir chapitre 4, a montré que seul certains cas particuliers supplémentaires sont couvert par « safe stack ». L'implémentation n'introduit pas de coût en performance significatif et est au même niveau des « stack cookies », les quelques pourcent de couverture supplémentaire sont donc bienvenus.

4 Proof of concept d'une attaque

L'implémentation de safe stack au sein de LLVM doit permettre de prévenir les attaques se basant sur l'exploitation d'un dépassement de tampon. Dans ce chapitre un « proof of concept » d'une telle attaque est décrit sur un exemple de code fictif. Afin de rendre plus portable et reproductible cette phase de test, un environnement spécifique est mis en place et décrit précisément ainsi que la recette utilisée lors de la phase de compilation.

L'objectif est de démontrer l'efficacité de Levee et de comparer l'approche avec celle des stack cookies. Pour ce faire, une première attaque a été pensée, malheureusement celle-ci n'avais pas la possibilité d'aboutir. Basé sur ses conclusions, une autre attaque a été mise en place. Leur description théorique ainsi que leurs implémentation sont décrites en détail.

Un des exemples d'attaque proposé dans ce chapitre est basé sur l'article « Introduction to return oriented programming (ROP) » du blog *codearcana.com* [36].

Contenu du chapitre

4.1 Environnement

Afin de faciliter la mise en place de l'attaque, la compilation est faite en 32 bits. L'environnement choisi pour installer la version 4.0 de LLVM est Debian 8 en 64 bits. Afin de pouvoir correctement compiler et exécuter en 32 bits, les bibliothèques nécessaires doivent être installés (lignes 20 à 22). En plus de LLVM, the GNU project debugger (GDB) ainsi que quelques autres utilitaires sont installés.

4.1.1 Docker

L'environnement décrit ci-dessus ainsi que l'installation des outils est mis en place grâce à Docker. Le Dockerfile suivant contient toutes les instructions nécessaire afin d'installer la version 4.0 de LLVM et Clang.

```
1  FROM debian:8
2  MAINTAINER "Joel.Gugger@master.hes-so.ch"
3
4  RUN apt-get update && \
5      apt-get upgrade -y && \
6      apt-get dist-upgrade -y
7
8  RUN apt-get install -y --no-install-recommends \
9          dialog apt-utils software-properties-common
10
11 # Install utilities
12 RUN apt-get install -y vim nano wget make
13
14 # Install llvm 4.0
15 RUN add-apt-repository 'deb http://apt.llvm.org/jessie/ llvm-toolchain-jessie-4.0 main'
16 RUN wget -O - http://apt.llvm.org/llvm-snapshot.gpg.key | apt-key add - && apt-get update
17 RUN apt-get install -y clang-4.0 lldb-4.0 lld-4.0
18
19 # Install 32 bits dependencies and gdb
20 RUN dpkg --add-architecture i386 && \
21         apt-get update && \
22         apt-get install -y build-essential gcc-multilib rpm libstdc++6:i386 libgcc1:i386
          ↪   zlib1g:i386 libncurses5:i386
23 RUN apt-get install -y gdb file
24
25 COPY ./.bash_profile /root/.bash_profile
26 RUN cat /root/.bash_profile >> /root/.bashrc
27
28 WORKDIR /shared
29
30 ENTRYPOINT echo "The container is ready!" && sleep 10000000
```

Listing 4.1 Fichier décrivant l'environnement choisi pour l'installation de LLVM 4 sous Debian 8

Il existe plusieurs manières d'installer LLVM. Il serait tout à fait possible de compiler directement depuis les sources. Plusieurs de ces techniques ont été testées, et la suivante a été retenue : l'installation de Clang 4.0, LLDB (équivalant de GDB) et de LLD (le « linker » de LLVM) se fait en rajoutant le dépôt APT de LLVM. Après plusieurs essais, le débogueur GDB est préféré à LLDB et est par la suite utilisé à la place de LLDB, ce dernier n'étant pas encore assez complet et globalement utilisé.

Tout les autres fichiers de configurations de l'environnement Docker sont disponibles à l'annexe A.

4.1.2 Gestion de la compilation

L'utilitaire *make* est utilisé pour gérer le processus de compilation. Un Makefile est mis en place afin de rassembler les différents « flags » utilisés pour générer les deux versions exécutables (l'un avec safe stack et l'autre sans). En en-tête, différentes variables utilisées par *make* sont définies afin de s'assurer que Clang 4.0 est bien utilisé. Deux version de l'exécutable sont créés : *safe-stack* et *stack-cookie*. Le language intermédiaire de LLVM est émit lors de la compilation afin de comparer les effets de l'un et l'autre mécanisme de protection.

À noté que la protection de la pile *-fstack-protector* est descativée grâce à *-fno-stack-protector* lorsque l'on active safe stack. Le but est de comparer les deux et actuellement, safe stack n'est pas compatible avec les stack cookies.

```
1  CC = clang-4.0
2  LL = llc-4.0
3  CFLAGS = -Wall -g -m32 -o1
4
5  # Safe Stack
6  SS_FLAG= -fsanitize=safe-stack -fno-stack-protector
7  # Stack Cookies
8  SC_FLAG= -fstack-protector
9
10 TARGET = struct.c
11 BUILDDIR = build
12
13 SAFE = safe-stack
14 NOSAFE = stack-cookie
15
16
17 all: dir $(BUILDDIR)/$(SAFE) $(BUILDDIR)/$(NOSAFE)
18
19 dir:
20         mkdir -p $(BUILDDIR)
21
22 $(BUILDDIR)/$(SAFE): $(BUILDDIR)/$(SAFE).ll
23         $(CC) $(CFLAGS) $(SS_FLAG) $? -o $@
24
25 $(BUILDDIR)/$(NOSAFE): $(BUILDDIR)/$(NOSAFE).ll
26         $(CC) $(CFLAGS) $(SC_FLAG) $? -o $@
27
28 $(BUILDDIR)/$(SAFE).ll: $(TARGET)
29         $(CC) -S -emit-llvm $(CFLAGS) $(SS_FLAG) -c $< -o $@
30
31 $(BUILDDIR)/$(NOSAFE).ll: $(TARGET)
32         $(CC) -S -emit-llvm $(CFLAGS) $(SC_FLAG) -c $< -o $@
33
34 clean:
35         rm -f $(BUILDDIR)/*~ $(BUILDDIR)/*.o $(BUILDDIR)/*.ll \
36         $(BUILDDIR)/*.s $(BUILDDIR)/$(SAFE) $(BUILDDIR)/$(NOSAFE)
37
38 check:
39         ./checksec.sh --file $(BUILDDIR)/$(SAFE)
40         ./checksec.sh --file $(BUILDDIR)/$(NOSAFE)
```

Listing 4.2 Makefile regroupant les différentes options de compilations

Un script permettant d'afficher les mécanismes de protections d'un binaire est utilisé, pour cela une règle spécifique aux tests est décrite à la ligne 38.

4.1.3 Mécanismes de sécurité actifs

ASLR est actif au sein de l'environnement. Chaque binaire est protegé soit avec stack cookies ou avec safe stack. Le script `checksec.sh` [37] est utilisé pour afficher les mécanismes actifs sur chaque binaire. On peut constater sur le résultat ci-dessous que les stack cookies sont bien desactivés lors du test de safe stack et que dans les deux cas le drapeau NX est actif.

```
root@5017cf36ad41:/shared/rop# make check
./checksec.sh --file build/safe-stack
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH
No RELRO        No canary found   NX enabled    No PIE          No RPATH   No RUNPATH

./checksec.sh --file build/stack-cookie
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH
No RELRO        Canary found      NX enabled    No PIE          No RPATH   No RUNPATH
```

Listing 4.3 Resultat du test de sécurité par checksec.sh

— PIE est un mécanisme de protection qui n'a pas été abordé dans le chapitre 2 mais qui pourrait y avoir une place. C'est un mécanisme récent (par rapport à ceux abordés dans le chapitre) permettant d'appliquer de l'aléatoire sur des régions de code, et ainsi rendre plus difficile la mise en place d'une attque de type ROP. —

4.2 Pointeur de fonction sur la pile

4.2.1 Description théorique de l'attaque

La première tentative se base sur une *structure* dans laquelle est stocké un pointeur vers une fonction. En placant la *structure* et le tampon sur la pile d'exécution, l'idée est de réécrir l'adresse du pointeur présent dans ladite *structure* afin de lancer la chaîne de gadgets à l'appel du pointeur.

Le but est de mettre en place une « stack frame » correspondant à la fonction `main()` ayant les variables locales stockées dans l'ordre dans lesquelles elles sont déclarées, comme le montre la Figure 4.1.

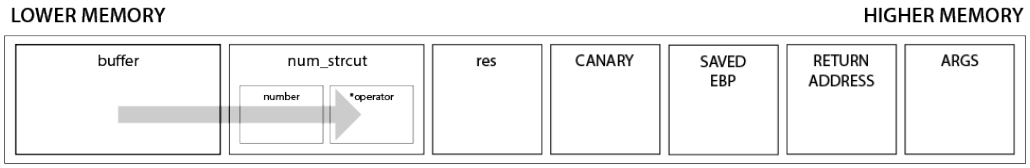

FIGURE *4.1 Résultat attendu de la « stack frame » sur la première attaque*
***Source:** Auteur*

4.2.2 Implémentation

Le code correspondant au scénario, présenté en Listing 4.4, comprend : (i) une structure nommée `Number`, (ii) une fonction `call()` permettant d'appliquer la fonction pointée par la structure et (iii) une fonction retournant le carré — trés mal optimisé — de la valeur passée en paramètre. La fonction `main()` initialise la structure sur la pile

ainsi que le `buffer`, puis copie dans le buffer l'argument `argv[1]` passé à l'exécution et appel la fonction `call()`. De manière thèorique il est alors possible, via l'arguement `argv[1]`, de réécrire l'adresse stockée dans la structure.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4
5  typedef struct Number {
6      int number;
7      int (*operator)();
8  } Number;
9
10 int call(struct Number self)
11 {
12     return self.operator(self.number);
13 }
14
15 int square(int number)
16 {
17     return number * number;
18 }
19
20 int main(int argc, char **argv)
21 {
22     int res;
23     Number num_strcut;
24     char buffer[20];
25
26     num_strcut.operator = &square;
27     num_strcut.number = 10;
28     strcpy(buffer, argv[1]);
29
30     res = call(num_strcut);
31     printf("%i\n", res);
32
33     return 0;
34 }
```

Listing 4.4 Source du premier scénario d'attaque

Cependant, lors que l'on inspect la pile d'exécution, on se rend compte que l'ordre dans lequel les variables sont stockées a été modifié. Le « buffer » est positionné en premier — vers la gauche —, ce qui rend impossible son exploitation. Ce comportement inatendu est le résultat d'une manipulation souhaitée par le compilateur. Afin de se prémunir au mieux contre l'exploitation des dépassements de tampon, il place les variables de type `char*` dans les adresses les plus hautes, juste après le canari. De cette manière, si un dépassement apparaît, il est tout de suite détecté par le canari.

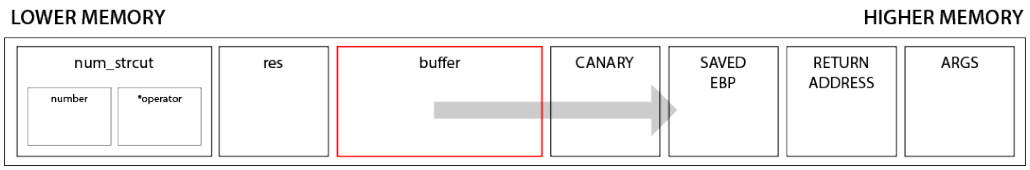

FIGURE *4.2 Résultat obtenu de la « stack frame » sur la première attaque*
***Source:** Auteur*

4.2.3 Conclusion

Cette attaque ne peux pas aboutir et est detectée avec les deux mécanismes de protection. Le but ici est de démontré les cas de figures où safe stack permet de se protéger contre des attaques qui ne seraient pas gardée par les stack cookies, cette attaque est donc abandonée au profit d'une nouvelle approche.

4.3 Bypass du canari

4.3.1 Description théorique de l'attaque

4.3.2 Implémentation

4.3.3 Conclusion

4.4 Conclusions

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 | Configuration du Docker

Le docker-compose s'occupe du démarage d'un ou plusieurs container ainsi que de leur interaction avec l'hôte (volumes, ports, etc). Un seul volume partagé est monté au sein du container, sous l'adresse `/shared`. Afin de pouvoir désactiver certaines protection tel que ASLR, il est nécessaire d'autorisé des privilèges suppérieurs au container (voir ligne 6).

```
1  version: '2.1'
2  services:
3    levee:
4      command: >
5          bash -c "echo Container ready! Sleep 100000000... && sleep 100000000"
6      privileged: true
7      container_name: levee
8      build: ./levee
9      image: levee/levee
10     volumes:
11         - ./levee/shared:/shared
```

Listing A.1 Fichier de configuration général utilisé par docker-compose

Le contenu d'un fichier `.bash_profile` est copié lors de la phase de contruction du container dans le dossier de l'utilisateur et est rajouté au fichier `.bashrc`, cela permet de rajouter facilement des commandes ou des raccourcis. Actuelement le fichier ne contient pas grand chose mais le méchanisme est en place.

```
alias ll="ls -lah --color"
```

Listing A.2 Fichier bash copier dans le .bashrc de l'utilisateur Debian

Le fichier principal contenant les instructions de construction de l'image est dispnible en Listing 4.1.

B Implémentation de « safe stack »

```
1  /// Check whether a given allocation must be put on the safe
2  /// stack or not. The function analyzes all uses of AI and checks whether it is
3  /// only accessed in a memory safe way (as decided statically).
4  bool SafeStack::IsSafeStackAlloca(const Value *AllocaPtr, uint64_t AllocaSize) {
5    // Go through all uses of this alloca and check whether all accesses to the
6    // allocated object are statically known to be memory safe and, hence, the
7    // object can be placed on the safe stack.
8    SmallPtrSet<const Value *, 16> Visited;
9    SmallVector<const Value *, 8> WorkList;
10   WorkList.push_back(AllocaPtr);
11
12   // A DFS search through all uses of the alloca in bitcasts/PHI/GEPs/etc.
13   while (!WorkList.empty()) {
14     const Value *V = WorkList.pop_back_val();
15     for (const Use &UI : V->uses()) {
16       auto I = cast<const Instruction>(UI.getUser());
17       assert(V == UI.get());
18
19       switch (I->getOpcode()) {
20       case Instruction::Load: {
21         if (!IsAccessSafe(UI, DL->getTypeStoreSize(I->getType()), AllocaPtr,
22                           AllocaSize))
23           return false;
24         break;
25       }
26       case Instruction::VAArg:
27         // "va-arg" from a pointer is safe.
28         break;
29       case Instruction::Store: {
30         if (V == I->getOperand(0)) {
31           // Stored the pointer - conservatively assume it may be unsafe.
32           DEBUG(dbgs() << "[SafeStack] Unsafe alloca: " << *AllocaPtr
33                        << "\n            store of address: " << *I << "\n");
34           return false;
35         }
36
37         if (!IsAccessSafe(UI, DL->getTypeStoreSize(I->getOperand(0)->getType()),
38                           AllocaPtr, AllocaSize))
39           return false;
40         break;
41       }
42       case Instruction::Ret: {
43         // Information leak.
44         return false;
45       }
46
47       case Instruction::Call:
48       case Instruction::Invoke: {
49         ImmutableCallSite CS(I);
50
51         if (const IntrinsicInst *II = dyn_cast<IntrinsicInst>(I)) {
52           if (II->getIntrinsicID() == Intrinsic::lifetime_start ||
53               II->getIntrinsicID() == Intrinsic::lifetime_end)
54             continue;
55         }
56
57         if (const MemIntrinsic *MI = dyn_cast<MemIntrinsic>(I)) {
58           if (!IsMemIntrinsicSafe(MI, UI, AllocaPtr, AllocaSize)) {
59             DEBUG(dbgs() << "[SafeStack] Unsafe alloca: " << *AllocaPtr
60                          << "\n            unsafe memintrinsic: " << *I
61                          << "\n");
62             return false;
63           }
64           continue;
65         }
66
```

```
67          // LLVM 'nocapture' attribute is only set for arguments whose address
68          // is not stored, passed around, or used in any other non-trivial way.
69          // We assume that passing a pointer to an object as a 'nocapture
70          // readnone' argument is safe.
71          // FIXME: a more precise solution would require an interprocedural
72          // analysis here, which would look at all uses of an argument inside
73          // the function being called.
74          ImmutableCallSite::arg_iterator B = CS.arg_begin(), E = CS.arg_end();
75          for (ImmutableCallSite::arg_iterator A = B; A != E; ++A)
76            if (A->get() == V)
77              if (!(CS.doesNotCapture(A - B) && (CS.doesNotAccessMemory(A - B) ||
78                                                 CS.doesNotAccessMemory()))) {
79                DEBUG(dbgs() << "[SafeStack] Unsafe alloca: " << *AllocaPtr
80                             << "\n            unsafe call: " << *I << "\n");
81                return false;
82              }
83          continue;
84        }
85
86        default:
87          if (Visited.insert(I).second)
88            WorkList.push_back(cast<const Instruction>(I));
89        }
90      }
91    }
92
93    // All uses of the alloca are safe, we can place it on the safe stack.
94    return true;
95  }
```

Listing B.1 Implémentation de la méthode IsSafeStackAlloca de « safe stack »

```
1   bool SafeStack::runOnFunction(Function &F) {
2     DEBUG(dbgs() << "[SafeStack] Function: " << F.getName() << "\n");
3
4     if (!F.hasFnAttribute(Attribute::SafeStack)) {
5       DEBUG(dbgs() << "[SafeStack]     safestack is not requested"
6                       " for this function\n");
7       return false;
8     }
9
10    if (F.isDeclaration()) {
11      DEBUG(dbgs() << "[SafeStack]     function definition"
12                      " is not available\n");
13      return false;
14    }
15
16    if (!TM)
17      report_fatal_error("Target machine is required");
18    TL = TM->getSubtargetImpl(F)->getTargetLowering();
19    SE = &getAnalysis<ScalarEvolutionWrapperPass>().getSE();
20
21    ++NumFunctions;
22
23    SmallVector<AllocaInst *, 16> StaticAllocas;
24    SmallVector<AllocaInst *, 4> DynamicAllocas;
25    SmallVector<Argument *, 4> ByValArguments;
26    SmallVector<ReturnInst *, 4> Returns;
27
28    // Collect all points where stack gets unwound and needs to be restored
29    // This is only necessary because the runtime (setjmp and unwind code) is
30    // not aware of the unsafe stack and won't unwind/restore it properly.
31    // To work around this problem without changing the runtime, we insert
32    // instrumentation to restore the unsafe stack pointer when necessary.
33    SmallVector<Instruction *, 4> StackRestorePoints;
34
35    // Find all static and dynamic alloca instructions that must be moved to the
```

```
36    // unsafe stack, all return instructions and stack restore points.
37    findInsts(F, StaticAllocas, DynamicAllocas, ByValArguments, Returns,
38              StackRestorePoints);
39
40    if (StaticAllocas.empty() && DynamicAllocas.empty() &&
41        ByValArguments.empty() && StackRestorePoints.empty())
42      return false; // Nothing to do in this function.
43
44    if (!StaticAllocas.empty() || !DynamicAllocas.empty() ||
45        !ByValArguments.empty())
46      ++NumUnsafeStackFunctions; // This function has the unsafe stack.
47
48    if (!StackRestorePoints.empty())
49      ++NumUnsafeStackRestorePointsFunctions;
50
51    IRBuilder<> IRB(&F.front(), F.begin()->getFirstInsertionPt());
52    UnsafeStackPtr = TL->getSafeStackPointerLocation(IRB);
53
54    // Load the current stack pointer (we'll also use it as a base pointer).
55    // FIXME: use a dedicated register for it ?
56    Instruction *BasePointer =
57        IRB.CreateLoad(UnsafeStackPtr, false, "unsafe_stack_ptr");
58    assert(BasePointer->getType() == StackPtrTy);
59
60    AllocaInst *StackGuardSlot = nullptr;
61    // FIXME: implement weaker forms of stack protector.
62    if (F.hasFnAttribute(Attribute::StackProtect) ||
63        F.hasFnAttribute(Attribute::StackProtectStrong) ||
64        F.hasFnAttribute(Attribute::StackProtectReq)) {
65      Value *StackGuard = getStackGuard(IRB, F);
66      StackGuardSlot = IRB.CreateAlloca(StackPtrTy, nullptr);
67      IRB.CreateStore(StackGuard, StackGuardSlot);
68
69      for (ReturnInst *RI : Returns) {
70        IRBuilder<> IRBRet(RI);
71        checkStackGuard(IRBRet, F, *RI, StackGuardSlot, StackGuard);
72      }
73    }
74
75    // The top of the unsafe stack after all unsafe static allocas are
76    // allocated.
77    Value *StaticTop =
78        moveStaticAllocasToUnsafeStack(IRB, F, StaticAllocas, ByValArguments,
79                                       Returns, BasePointer, StackGuardSlot);
80
81    // Safe stack object that stores the current unsafe stack top. It is updated
82    // as unsafe dynamic (non-constant-sized) allocas are allocated and freed.
83    // This is only needed if we need to restore stack pointer after longjmp
84    // or exceptions, and we have dynamic allocations.
85    // FIXME: a better alternative might be to store the unsafe stack pointer
86    // before setjmp / invoke instructions.
87    AllocaInst *DynamicTop = createStackRestorePoints(
88        IRB, F, StackRestorePoints, StaticTop, !DynamicAllocas.empty());
89
90    // Handle dynamic allocas.
91    moveDynamicAllocasToUnsafeStack(F, UnsafeStackPtr, DynamicTop,
92                                    DynamicAllocas);
93
94    // Restore the unsafe stack pointer before each return.
95    for (ReturnInst *RI : Returns) {
96      IRB.SetInsertPoint(RI);
97      IRB.CreateStore(BasePointer, UnsafeStackPtr);
98    }
99
100   DEBUG(dbgs() << "[SafeStack]     safestack applied\n");
101   return true;
102 }
```

Listing B.2 Implémentation de la méthode runOnFunction de « safe stack »

```
97  static inline void *unsafe_stack_alloc(size_t size, size_t guard) {
98    CHECK_GE(size + guard, size);
99    void *addr = MmapOrDie(size + guard, "unsafe_stack_alloc");
100   MprotectNoAccess((uptr)addr, (uptr)guard);
101   return (char *)addr + guard;
102 }
103
104 static inline void unsafe_stack_setup(void *start, size_t size, size_t guard) {
105   CHECK_GE((char *)start + size, (char *)start);
106   CHECK_GE((char *)start + guard, (char *)start);
107   void *stack_ptr = (char *)start + size;
108   CHECK_EQ((((size_t)stack_ptr) & (kStackAlign - 1)), 0);
109
110   __safestack_unsafe_stack_ptr = stack_ptr;
111   unsafe_stack_start = start;
112   unsafe_stack_size = size;
113   unsafe_stack_guard = guard;
114 }
115
116 static void unsafe_stack_free() {
117   if (unsafe_stack_start) {
118     UnmapOrDie((char *)unsafe_stack_start - unsafe_stack_guard,
119                unsafe_stack_size + unsafe_stack_guard);
120   }
121   unsafe_stack_start = nullptr;
122 }
```

Listing B.3 Méthodes présentes dans la bibliothèque d'exécution de « safe stack »

Table des figures

Liste des codes sources

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*. 27 jan. 2009. URL : http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/.

[4] Gustavo Duarte. *How the Kernel Manages Your Memory*. 3 fév. 2009. URL : http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory/.

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

[6] *64-bit computing*. 27 avr. 2017. URL : https://en.wikipedia.org/wiki/64-bit_computing.

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

[8] *Data Execution Prevention*. 9 juin 2015. URL : https://fr.wikipedia.org/wiki/Data_Execution_Prevention.

[9] *PowerPC*. 29 sept. 2016. URL : https://fr.wikipedia.org/wiki/PowerPC.

[10] *Executable space protection*. 8 jan. 2017. URL : https://en.wikipedia.org/wiki/Executable_space_protection.

[11] *NX Bit*. 21 mar. 2017. URL : https://fr.wikipedia.org/wiki/NX_Bit.

[12] *Return-to-libc attack*. 7 juin 2016. URL : https://fr.wikipedia.org/wiki/Return-to-libc_attack.

[13] *Address space layout randomization [FR]*. 15 fév. 2016. URL : https://fr.wikipedia.org/wiki/Address_space_layout_randomization.

[14] *Address space layout randomization [EN]*. 28 mar. 2017. URL : https://en.wikipedia.org/wiki/Address_space_layout_randomization.

[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*. 11 oct. 2015. URL : http://taishi8117.github.io/2015/11/11/stack-bof-2/.

[17] *Position-independent executables*. 20 avr. 2017. URL : https://en.wikipedia.org/wiki/Position-independent_code#Position-independent_executables.

[18] *Stack canaries*. 30 mar. 2017. URL : https://en.wikipedia.org/wiki/Stack_buffer_overflow#Stack_canaries.

[19] *Sentinel species*. 16 juin 2016. URL : https://en.wikipedia.org/wiki/Sentinel_species#Historical_examples.

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

[21] *Buffer overflow protection*. 24 jan. 2017. URL : https://en.wikipedia.org/wiki/Buffer_overflow_protection#Canaries.

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

[23] 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.

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

[25] Artem DINABURG. *Let's talk about CFI : clang edition*. 17 oct. 2016. URL : https://blog.trailofbits.com/2016/10/17/lets-talk-about-cfi-clang-edition/.

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

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

[28] 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.

[29] 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.

[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] *Dependable Systems Lab*. 23 avr. 2017. URL : http://dslab.epfl.ch/.

[32] 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.

[33] *setjmp.h*. 30 déc. 2016. URL : https://en.wikipedia.org/wiki/Setjmp.h.

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

[35] *Chris Lattner's Homepage*. URL : http://nondot.org/~sabre/.

[36] Alex REECE. *Introduction to return oriented programming (ROP)*. 28 mai 2013. URL : http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html.

[37] Tobias KLEIN. *checksec.sh*. 2011. URL : `http://www.trapkit.de/tools/checksec.html`.

Glossaire

ASLR Address Space Layout Randomization, technique permettant de rendre aléatoire la position des ségments mémoires. 1, 10, 11, 15, 19, 32, 37

AST abstract syntax tree. 21

CFG control-flow graph. 13

CFI control-flow integrity. 1, 13, 15

Clang Clang est un compilateur pour les langages de programmation C, C++, Objective-C et Objective-C++. Son interface de bas niveau utilise les bibliothèques LLVM pour la compilation. 12, 13, 15, 20, 23, 25, 30, 31

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. 16–21, 25, 27

CPS code-pointer separation. 19–21

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

GDB the GNU project debugger. 30

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, 20, 23, 25, 29

LLDB LLDB Debugger est présenté comme étant un débegeur de dernière génération et éfficace. Il fait parti des outils développés au sein du plus large projet qu'est LLVM. 22, 30

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, 20–25, 27, 29–31, 45

LLVM IR LLVM intermediate representation. 22, 23, 25, 45

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, 32

PIE position-independent executable. 11, 32

ROP return oriented programming. 11, 14, 15, 32

safe stack Safe Stack est un composant de de CPI/CPS permettant de traiter particulièrement la gestion des pointeurs présents au sein de la pile d'exécution. 20, 25–27, 29, 31, 32, 34, 42, 45

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, 20, 27, 29, 31, 32, 34