UNIVERSITÉ
LAVAL
FACULTÉ DES SCIENCES ET DE GÉNIE
Département d'informatique et génie logiciel
Pavillon Adrien-Pouliot, local 3908
Cité universitaire
Québec, Canada
G1K 7P4

LES DÉBORDEMENTS DE TAMPONS ET LES VULNÉRABILITÉS DE CHAÎNE DE FORMAT

PAR

PATRICE LACROIX

RAPPORT DE RECHERCHE
DIUL-RR-0304

DÉPARTEMENT D'INFORMATIQUE ET DE GÉNIE LOGICIEL
FACULTÉ DES SCIENCES ET DE GÉNIE

Pavillon Adrien-Pouliot
Université Laval
Sainte-Foy, Québec, Canada
G1K 7P4

JUILLET 2003

Les débordements de tampons et les vulnérabilités de chaîne de format[1]

Patrice Lacroix

Groupe LSFM
Département d'informatique et de génie logiciel
Université Laval
Québec, QC G1K 7P4 Canada

18 décembre 2002
Révisé le 7 juillet 2003

[1]Cette recherche a pu être menée à terme grâce à une bourse de premier cycle du Conseil de recherche en sciences naturelles et en génie du Canada.

Table des matières

Liste des figures

Liste des exemples

Résumé

Les débordements de tampons sont connus depuis longtemps, mais encore aujourd'hui ils posent des problèmes de sécurité majeurs. Ils peuvent dans plusieurs cas être exploités pour rendre inactive la sécurité des logiciels. Les vulnérabilités liées aux chaînes de format quant à elles sont connues depuis bien moins longtemps, mais elles présentent un danger tout aussi grand pour la sécurité des systèmes.

Ce rapport étudie ces deux types de vulnérabilité, les techniques utilisées pour les exploiter de façon malveillante ainsi que les différentes approches qui permettent de les éviter ou de limiter leurs conséquences.

Chapitre 1

Introduction

1.1 Motivation

Depuis plusieurs années, les débordements de tampons posent des problèmes de sécurité majeurs. Ils sont la cause d'un grand pourcentage des vulnérabilités récentes. Selon les sources d'informations et les dates, jusqu'à 50% des vulnérabilités rapportées peuvent être causées par des débordements de tampons [Wag00, CWP+99]. Jusqu'en juin 2000, les débordements de tampons étaient encore la principale forme d'intrusion à distance utilisée par les pirates. Depuis ce moment, ce sont les vulnérabilités liées aux chaînes de format, connues publiquement depuis beaucoup moins longtemps, qui les ont dépassés à ce titre [CBB+01].

Dans l'optique où l'on veut écrire des programmes sécuritaires, il apparaît important d'en savoir plus sur les débordements de tampons et les vulnérabilités liées aux chaînes de format.

1.2 Portée

Ce rapport examine les deux classes de vulnérabilité mentionnées précédemment surtout du point de vue du code source écrit en C ou C++. Nous ne nous limitons cependant pas à ces langages et nous en touchons d'autres à l'occasion, par exemple Java, et même le code machine. Il faut noter que dans ce rapport, à moins d'avis contraire, on peut considérer que tout ce qui est décrit s'applique au langage C et que les programmes de démonstration ont été compilés avec les options par défaut de GCC version 2.95.4 sous Linux.

1.3 Débordements de tampons

Selon le Grand dictionnaire terminologique[1], un tampon est défini comme :

[1] `http://www.granddictionnaire.com/`

> Mémoire utilisée pour le stockage temporaire de données lors du transfert d'informations afin de compenser la différence de débit, de vitesse de traitement ou de synchronisation entre les divers dispositifs d'un ordinateur et ses périphériques.

Habituellement, lorsqu'on parle d'un débordement de tampon, on réfère à l'écriture passée la fin d'un tampon. Pour les fins de ce travail, nous utilisons l'expression *débordement de tampon* d'une façon plus large, de manière à couvrir tous les accès à la mémoire à l'extérieur des bornes d'un tableau, peu importe la façon dont il est défini, alloué ou utilisé. On parle donc de débordement de tampon aussi bien lorsqu'il est question de lecture que d'écriture.

Nous nous intéressons plus particulièrement aux cas où un débordement de tampon donne lieu à une vulnérabilité qui peut être exploitée de façon malveillante. Il n'est cependant pas très utile de faire cette distinction pour les raisons suivantes :

1. Pour vérifier qu'un débordement de tampon est exploitable, il faut d'abord savoir qu'il existe. Or, il est justement très difficile de déterminer si un débordement de tampon est possible dans un programme, comme nous allons le voir dans ce rapport.
2. Une fois qu'on a identifié une possibilité de débordement de tampon dans un programme, il est généralement très facile de le corriger.
3. À l'opposé, déterminer si un débordement de tampon est exploitable ou pas est beaucoup plus complexe, et dépend du compilateur utilisé et parfois même des options de compilation et d'édition des liens car les langages comme le C et le C++ ne définissent pas ce qui doit se passer lors d'un débordement de tampon.
4. Même si un débordement de tampon ne peut pas être exploité de façon malveillante, il peut tout de même avoir des impacts sur l'exécution d'un programme, ce qui n'est pas souhaitable.

Quoi qu'il en soit, nous verrons que plusieurs approches pour solutionner le problème des débordements de tampons ne s'attardent qu'à protéger certains cas plus susceptibles d'être vulnérables, mais sans nécessairement prouver qu'une vulnérabilité existe vraiment.

1.4 Vulnérabilités de chaîne de format

Une chaîne de format est le mécanisme utilisé par plusieurs fonctions normalisées de la bibliothèque du langage C pour créer une chaîne de caractères à partir de constantes et de variables. Les fonctions qui prennent en entrée une chaîne de format ont besoin d'un nombre variable de paramètres pour accompagner la chaîne de format. Ces arguments sont utilisés à mesure que des symboles à remplacer sont lus dans la chaîne de format.

Il y a une vulnérabilité lorsqu'un attaquant peut manipuler la chaîne de format dans le but d'obtenir des données auxquelles il ne devrait pas avoir accès ou de modifier le cours de l'exécution du programme de façon à ce qu'il exécute du code arbitraire ou malveillant.

Les vulnérabilités liées aux chaînes de format sont généralement classées dans une catégorie différente des débordements de tampons. Cependant, il paraît naturel d'associer le passage d'un nombre variable d'arguments au passage d'un tableau de longueur variable. Plusieurs des vulnérabilités liées aux chaînes de format sont causées par la lecture d'une valeur en dehors des bornes du tableau qui représente les paramètres variables. Nous sommes donc portés à croire que certaines techniques de

protection contre ces deux classes de vulnérabilité pourraient être similaires. Nous les étudions donc ensemble dans ce rapport.

1.5 Plan du rapport

Le chapitre 2 examine les différentes causes et circonstances qui peuvent mener à des débordements de tampons et à des vulnérabilités de chaîne de format. Les chapitres 3 et 4 étudient les conséquences que peuvent avoir ces vulnérabilités, particulièrement celles qui ont des répercussions sur la sécurité des systèmes. Le chapitre 5 étudie les différentes approches pour solutionner les problèmes engendrés par ces vulnérabilités. Le chapitre 6 présente notre conclusion.

L'annexe A présente en détails une attaque qui exploite une vulnérabilité de chaîne de format. L'annexe B présente certains logiciels qui peuvent être utiles dans le but de détecter ou d'éviter les débordements de tampons, les vulnérabilités de chaîne de format ou leurs conséquences.

Chapitre 2

Causes des vulnérabilités

Lorsqu'on utilise un langage comme le C, il est très facile d'écrire un programme qui contient involontairement des possibilités de débordement de tampon. En fait, on peut même dire qu'il est plutôt difficile d'écrire un programme C avec l'assurance qu'il ne contient aucun débordement de tampon possible [Wag00, p. 11]. Plusieurs facteurs contribuent à ceci, notamment le fait que le langage n'offre pas de sûreté de typage, qu'il inclut des fonctions normalisées qui sont très difficiles à utiliser de manière sûre, et une certaine culture de laisser-faire chez les programmeurs C.

2.1 Sûreté de typage

Cette section est basée sur [CPM+98, CWP+99, Rit93, MV00].

Certains langages offrent la sûreté de typage et sont par conséquent immunisés contre les débordements de tampons et les vulnérabilités de chaîne de format. Ce n'est évidement pas le cas pour le C et le C++. Un langage qui offre la sûreté de typage ne permet pas d'utiliser une variable d'une façon qui n'est pas compatible avec son véritable type. En conséquence, un tel langage ne peut pas permettre l'accès à un élément situé à l'extérieur des bornes d'un tableau puisque cet élément n'existe pas et ne peut donc pas être utilisé de manière compatible avec son véritable type. Pour les mêmes raisons il ne peut pas non plus permettre d'accéder à un paramètre d'une fonction qui n'aurait pas été passé.

Les langages C et C++ permettent des constructions qui vont à l'encontre d'un typage sûr. Par exemple, le programme de l'exemple 2.1 compile sans erreurs, ni avertissements avec la plupart des compilateurs C. Ce programme comporte deux problèmes de typage. D'abord il effectue le trans-

```
int main()
{
        *(char*)12345 = 'A';
        return 0;
}
```

Exemple 2.1: Démonstration de problèmes de typage

typage de l'entier 12345 vers le type pointeur de caractère (`char *`), alors que 12345 ne représente pas un pointeur. Selon la spécification des langages C et C++, transtyper un entier arbitraire vers un pointeur donne un résultat qui dépend du fabricant du compilateur. Quoi qu'il en soit, la plupart des implémentations de ces langages C donnent comme résultat un pointeur qui a la même représentation interne que l'entier, et qui pointe donc vers l'adresse mémoire 12345 du programme.

Le deuxième problème de typage consiste à déréférencer le pointeur nouvellement créé pour affecter le caractère `'A'` à la variable pointée. Puisqu'une zone de mémoire non définie ne possède pas de type, y accéder est toujours une erreur de typage. Dans notre exemple, il n'y a aucune façon d'être certain que la variable pointée est effectivement de type `char`. La variable se situe peut-être à une adresse qui ne fait même pas partie du programme. C'est cette dernière hypothèse qui se confirme lorsque nous exécutons le programme sous Linux après l'avoir compilé avec GCC.

Exécution de l'exemple 2.1

```
$ ./typage1
Erreur de segmentation
```

Le message `Erreur de segmentation` signifie que le programme a accédé à une adresse mémoire qui ne lui est pas allouée.

L'exemple 2.2 montre l'utilisation d'un indice en dehors des bornes d'un tableau. Il faut remar-

```
#include <stdio.h>

int main()
{
        /* Initialisation d'un tableau de 4 entiers */
        int A[4] = {1,2,3,4};
        /* Initialisation d'un tableau de 4 caractères nuls */
        char B[4] = {0};
        printf("avant: %d\n", A[0]);
        /* Affectation au sixième élément du tableau */
        B[5] = '9';
        printf("après: %d\n", A[0]);
        return 0;
}
```

Exemple 2.2: Problème de typage sur l'accès à un tableau

quer que les éléments des tableaux en C sont numérotés à partir de 0. L'indice 5 est donc le sixième élément du tableau. Il s'agit donc encore une fois d'une erreur de typage car le langage C ne définit pas ce qui est situé à l'extérieur des bornes d'un tableau. Si nous exécutons le programme après l'avoir compilé avec GCC, toujours sans avertissement, nous obtenons le résultat suivant :

Exécution de l'exemple 2.2

```
$ ./typage2
avant: 1
après: 14593
```

Ici nous pouvons expliquer ce qui c'est passé. Le tableau A était situé en mémoire juste après le tableau B, c'est pourquoi la valeur de son premier élément a été modifiée par l'affectation à B[5]. Nous pourrions aussi expliquer pourquoi la valeur de A[0] est 14593[1] mais ce qui est important de souligner est que la spécification du langage ne définit pas ce qui va se passer. Le résultat dépend donc du compilateur utilisé et on ne peut s'y fier lorsqu'on veut écrire des programmes portables. En fait le résultat dépend aussi de l'organisation interne de la mémoire dans l'ordinateur et il serait différent même si on utilisait la même version du compilateur GCC[2], mais pour un processeur PowerPC qui place les entiers en mémoire d'une façon différente.

L'exemple 2.3 montre des erreurs de typage avec l'utilisation d'une chaîne de format. Le pro-

```
#include <stdio.h>

int main()
{
        char c = 1;
        short s = 2;
        int i = 3;
        long l = 4;
        printf("%d, %d, %d, %d, %d\n", c, s, i, l, 5);
        return 0;
}
```

Exemple 2.3: Problème de typage avec une chaîne de format

gramme appelle la fonction `printf()` qui prend en paramètre une chaîne de format et un nombre variable d'arguments. Les problèmes de typage sont dus au fait qu'on passe comme argument des variables de types différents et qu'on demande à la fonction `printf()` de toutes les considérer comme des `int`. Si nous exécutons le programme, nous obtenons le résultat suivant :

Exécution de l'exemple 2.3

```
$ ./typage3
1, 2, 3, 4, 5
```

Les cinq paramètres sont affichés correctement malgré que la chaîne de format ne spécifie le bon type que pour la variable i. Nous observons donc que les erreurs de typage n'entraînent pas toujours

[1] Indice : $14593 = 1 + 9_{\text{ascii}} \times 2^8$.

[2] Le compilateur GCC permet de générer du code pour des dizaines de processeurs et de systèmes d'exploitation différents.

des comportements non souhaités. Ici, si tout se passe bien, c'est parce que GCC utilise le même format interne lors du passage en paramètre de tous les types de données utilisés dans l'exemple.

Il faut aussi noter que cette fois GCC donne un avertissement pour la non-concordance de type entre le paramètre de remplacement «`%d`» et la variable *l*, mais seulement à la condition d'activer la classe d'avertissement `format` avec les options `-Wformat` ou `-Wall`. En ce qui concerne les erreurs de typage des autres paramètres, le compilateur reste muet.

À l'opposé du C et du C++, Java est un langage avec un typage sûr, malgré qu'il ait une syntaxe qui ressemble à celle du C. Pour en arriver à avoir un typage sûr, Java a sacrifié certaines constructions qui existent dans le langage C, par exemple, les pointeurs et les fonctions avec un nombre d'arguments variable. Des pointeurs sont utilisés à l'intérieur de la machine virtuelle pour représenter les références aux objets du langage, mais ces pointeurs sont invisibles pour le programmeur Java et, surtout, il est impossible de les manipuler de façon arbitraire en utilisant des conversions de type ou en effectuant des opérations arithmétiques sur les pointeurs.

Ces restrictions règlent donc le cas des vulnérabilités liées aux chaînes de format, mais qu'en est-il du dépassement des bornes lors de l'accès à un tableau ? Considérons le programme de l'exemple 2.4 qui essaie d'accéder à un élément à l'extérieur des bornes du tableau. Java ne tente pas d'identifier

```
class Debordement1 {
    public static void main(String arg[])
    {
        int a[] = new int[4];
        a[4] = 4;  // a[4] est le cinquième élément!
    }
}
```

Exemple 2.4: Débordement de tableau en Java

ces situations à la compilation, ce qui serait impossible à faire précisément pour le cas général. Par contre, il détecte ces erreurs lors de l'exécution et, au lieu de donner un résultat indéterminé comme c'est le cas en C, la spécification du langage Java indique qu'une exception doit être lancée. Cette exception peut soit être attrapée par le programme qui peut alors prendre les mesures qui s'imposent, soit être ignorée ce qui a pour conséquence d'arrêter le fil d'exécution[3]. C'est ce dernier cas qui se produit dans notre exemple.

Exécution de l'exemple 2.4

```
$ java Debordement1
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
        at Debordement1.main(t.java:5)
```

S'il existe des langages avec un typage sûr, on peut se demander pourquoi le langage C n'offre pas un typage sûr. L'explication se trouve dans les origines du langage [Rit93]. Il faut savoir que le langage C a été développé en tant que langage de haut niveau permettant l'implémentation du système d'exploitation UNIX. On avait besoin d'un langage qui rivalise avec l'assembleur par sa

[3]Fil d'exécution est la traduction du terme anglais *thread*.

performance. C'est donc pour avoir une plus grande vitesse d'exécution qu'il n'y a pas de vérification automatique des bornes des tableaux. Le programmeur doit s'assurer lui-même que les bornes des tableaux soient respectées.

Étant donné qu'un système d'exploitation doit interagir étroitement avec l'ordinateur sur lequel il s'exécute, le transtypage entre entiers et pointeurs et l'arithmétique sur les pointeurs permettent d'accéder au matériel de façon relativement simple et performante.

Les erreurs présentées jusqu'ici sont évidentes pour ceux qui connaissent les langages discutés. Elles ont été présentées pour mettre en évidence l'importance du typage. Les sections qui suivent donnent des exemples d'erreurs qui sont plus susceptibles de se produire dans des vrais programmes.

2.2 Tampon trop petit

Cette section est basée sur [One96, Smi97].

Dans certains cas, un programmeur définit un tampon de taille fixe pour accueillir les données d'une source quelconque. Ceci est fréquent lorsqu'il sait que les entrées valides à un certain endroit dans le programme ont une taille maximale. C'est le cas, par exemple, lorsqu'on demande à l'utilisateur d'effectuer un choix parmi les options d'un menu et que chacun des choix s'exprime par une seule lettre.

Il est cependant possible que les données entrées ne soient pas valides. Dans ces cas, si le programmeur n'est pas prudent et qu'il ne valide pas la taille des données en entrée, il risque d'y avoir des débordements de tampons.

En C, le problème est amplifié par le fait que les chaînes de caractères sont délimitées par des caractères nuls plutôt que par des variables séparées qui indiquent la longueur. Le programmeur a donc généralement deux conditions à vérifier dans les boucles sur les caractères d'une chaîne, une pour surveiller le caractère nul et une autre pour surveiller les bornes du tampon. Par négligence, il peut omettre une des deux conditions, comme c'est le cas dans l'exemple 2.5.

Dans cet exemple, *chaine* représente une chaîne de caractères de longueur variable qui a été obtenue en entrée. On a défini le tableau *maj* avec une taille fixe pour y copier la chaîne en la transformant en majuscule. On omet de vérifier qu'on ne dépasse pas la capacité de *maj* ce qui provoque un débordement de tampon puisque la chaîne est trop grande pour le tampon.

Une alternative serait d'allouer dynamiquement la mémoire requise pour la chaîne *maj* en fonction de la longueur de *chaine*. Il faut garder à l'esprit qu'il n'y a pas de mal à utiliser un tampon avec une taille fixe. Il y a des raisons légitimes pour traiter une quantité maximum de données, surtout lorsqu'elles proviennent d'une source inconnue. Par contre, lorsqu'on utilise un tampon de taille fixe, il ne faut pas oublier de vérifier les bornes du tableau.

2.3 Interface sans vérification des bornes

La bibliothèque du C comprend des fonctions normalisées ayant une interface ne leur permettant pas d'effectuer la vérification des bornes des tampons. Voici un résumé de ces fonctions qui

```
#include <ctype.h>

int main()
{
        /* La chaîne qui suit a plus de 100 caractères */
        unsigned char *chaine =
                        "Quelqu'un avec des intentions "
                        "malveillantes pourrait s'organiser "
                        "pour que cette chaîne de "
                        "caractères soit beaucoup plus "
                        "longue que ce que le programmeur "
                        "a prévu traiter.";
        static unsigned char maj[100];
        int i;
        /* La boucle transforme chaine en majuscules */
        for (i=0; chaine[i]!=0; i++)
                maj[i] = toupper(chaine[i]);
        return 0;
}
```

Exemple 2.5: Dépassement des bornes d'un tampon. Le programmeur a omis de vérifier les bornes du tableau dans la condition d'arrêt de la boucle.

sont considérées dangereuses, soit parce qu'elles ne permettent pas la vérification des bornes, soit parce qu'elles ne l'obligent pas. Elles sont regroupées selon la particularité qui les rend dangereuses. Lorsqu'il existe des fonctions équivalentes mais moins dangereuses, elles sont indiquées avec leurs différences. Les comportements des fonctions décrites sont ceux de la version 2.2 de glibc[4], la bibliothèque C qu'on retrouve le plus souvent sous Linux.

Cette section est basée sur [Wag00, Smi97, VBKM01, Whe02c, MV00].

2.3.1 strcpy() et wcscpy()

Les fonctions `strcpy()` et `wcscpy()` prennent en paramètre un tampon et une chaîne. `strcpy()` agit sur des chaînes de caractères de type `char` tandis que `wcscpy()`[5] agit sur des chaînes de caractères de type `wchar_t`. Lorsqu'on les appelle, on doit s'assurer que le tampon est au moins aussi grand que la chaîne.

Dans l'exemple 2.6, *chaine* représente encore une fois une chaîne de caractères de longueur variable qui a été obtenue en entrée. Il y a un débordement de tampon car aucune vérification n'est faite pour s'assurer que la taille de la chaîne et du tampon concordent. On peut régler le problème en allouant suffisamment de mémoire de façon dynamique, par exemple avec `strlen()` et `malloc()`.

Une alternative à l'utilisation des fonctions `strcpy()` et `wcscpy()` est de les remplacer par `strncpy()` et `wcsncpy()` respectivement. Ces dernières prennent un paramètre supplémentaire, soit

[4] glibc est le nom usuel pour «GNU C Library».
[5] wcs vient de *wide character string*.

```
#include <string.h>

int main()
{
        /* La chaîne qui suit a plus de 100 caractères */
        unsigned char *chaine =
                        "Quelqu'un avec des intentions "
                        "malveillantes pourrait s'organiser "
                        "pour que cette chaîne de "
                        "caractères soit beaucoup plus "
                        "longue que ce que le programmeur "
                        "a prévu traiter.";
        static unsigned char tampon[100];
        /* L'instruction suivante fait déborder le tampon */
        strcpy(tampon, chaine);
        return 0;
}
```

Exemple 2.6: Débordement d'un tampon avec la fonction `strcpy()`. Le programmeur ne s'assure pas que le tampon est suffisamment grand pour contenir la chaîne à copier.

Il suffit de remplacer la déclaration du tampon de l'exemple 2.6

```
        static unsigned char tampon[100];
```

par une allocation dynamique comme la suivante :

```
        unsigned char *tampon;
        tampon = (unsigned char *)malloc(strlen(chaine)+1);
```

Exemple 2.7: Tampon alloué dynamiquement pour éviter les débordements

la taille du tampon, et elles ne dépassent jamais la taille qu'on leur spécifie. Elles ont cependant deux inconvénients. D'abord il peut y avoir des problèmes de performance si la taille du tampon est beaucoup plus grande que la chaîne à copier. Ceci est dû au fait que suite à la copie de la chaîne, la partie inutilisée du tampon est remplie par des caractères nuls.

Ensuite, la chaîne risque de ne pas se terminer par un caractère nul. Ce problème est beaucoup plus important du point de vue de la sécurité car en C, une chaîne de caractères doit habituellement être terminée par un caractère nul. Si elle ne l'est pas, le programme risque de dépasser les bornes du tampon en cherchant le caractère nul. Des données qui proviennent d'une autre variable peuvent «apparaître» dans la chaîne copiée, ce qui pourrait être grave s'il s'agit d'un mot de passe ou d'autres données confidentielles. Si on utilise `strncpy()`, il faut donc toujours écraser le dernier caractère du tampon par un caractère nul, comme le fait l'exemple 2.8.

```
char tampon[10];
char *chaine = "Bonjour tout le monde!";
strncpy(tampon, chaine, 10);
/* Caractère nul pour terminer la chaîne */
tampon[9] = '\0';
```

Exemple 2.8: Utilisation de `strncpy()`

2.3.2 strcat() et wcscat()

Les fonctions `strcat()` et `wcscat()` ressemblent respectivement à `strcpy()` et `wcscpy()`. La chaîne est cependant copiée à la suite de la chaîne qui se trouve déjà dans le tampon plutôt qu'à son début. Ceci signifie que le tampon doit avoir une taille assez grande pour recevoir la chaîne qu'il contient déjà plus la chaîne à copier.

Elles possèdent également des alternatives qui sont `strncat()` et `wcsncat()`. Ces fonctions n'ont pas le problème de performance de `strncpy()` et `wcsncpy()`. Elles diffèrent aussi de ces dernières par le fait que le caractère nul est toujours ajouté à la fin de la chaîne. Enfin, la taille qu'on passe à `strncat()` n'est pas celle du tampon, mais le nombre de caractères maximal à copier de la chaîne. Dans le calcul de la taille du tampon requise, il faut donc compter la longueur de la chaîne qui se trouve déjà dans le tampon, le nombre maximal de caractères à copier qui est passé en paramètre et additionner un pour le caractère nul qui est toujours ajouté.

2.3.3 sprintf() et vsprintf()

Les fonctions `sprintf()` et `vsprintf()` sont plus difficiles à utiliser correctement car elles prennent en paramètre une chaîne de format qui sert à formater le contenu d'un tampon. Cette chaîne peut contenir des spécificateurs de conversion à remplacer par des valeurs de longueur variable. Par exemple, un «`%s`» est remplacé par une chaîne de caractères. Il faut donc prendre en considération la longueur de cette dernière lorsqu'on calcule la taille requise pour le tampon.

Une alternative est de spécifier une «précision» avec le spécificateur de conversion pour une chaîne de caractères. De cette façon, seulement les premiers caractères de la chaîne sont copiés. On

peut donner la précision soit directement dans la chaîne de format ou la passer dans une variable en paramètre. L'exemple 2.9 montre les deux façons de faire.

```
char tampon[10];
sprintf(tampon, "%.9s", "Bonjour tout le monde!");
sprintf(tampon, "%.*s", 9, "Bonjour tout le monde!");
```

Exemple 2.9: «Précision» d'une chaîne de caractères avec sprintf

Il est très important de remarquer que la spécification de précision s'effectue après le point. Si on enlève le point, la valeur devient la taille minimale du champ, et il n'y a alors aucune protection contre les débordements. Lorsqu'on utilise la «précision» pour limiter la longueur d'une chaîne, il ne faut pas oublier de prendre en compte le reste de la chaîne de format dans le calcul de la taille de tampon requise. En particulier, il ne faut pas oublier le caractère nul.

Un «`%d`» peu aussi prendre une longueur variable une fois formaté. En supposant qu'il n'y a pas de contraintes supplémentaires, la longueur maximale varie en fonction de la taille d'un `int` qui elle-même varie en fonction du compilateur et du processeur. Elle sera de 6 caractères pour des `int` de 16 bits, 11 caractères pour des `int` de 32 bits et 20 caractères pour des `int` de 64 bits.

De plus, il ne faut pas oublier que la chaîne de format peut contenir plusieurs paramètres de remplacement. Ceci complexifie le calcul de la taille nécessaire pour le tampon et fait en sorte que des erreurs peuvent se glisser dans ce calcul. Si une erreur de calcul fait en sorte que le tampon réservé est trop petit, il risque d'y avoir un débordement. Il faut donc être très vigilent lorsqu'on choisit la taille du tampon servant à accueillir la chaîne formatée.

Les fonctions `snprintf()` et `vsnprintf()` sont des fonctions équivalentes à `sprintf()` et `vsprintf()` respectivement, mais elles sont généralement considérées moins dangereuses car elles acceptent comme paramètre la taille du tampon et elles la respectent. Contrairement à `strncpy()`, ces fonctions réservent toujours de l'espace pour un caractère nul à la fin du tampon. Évidemment que pour éviter les débordements avec ces fonctions, il faut que la taille qu'on passe à la fonction ne dépasse pas la taille réelle du tampon. Une chose qui peut nuire à l'utilisation de ces fonctions est qu'elles ne sont pas disponibles sur tous les systèmes. Elles le sont sous Linux, mais leur absence peut causer des problèmes lorsque vient le temps d'écrire du code qui doit fonctionner sur des systèmes différents.

2.3.4 scanf() et ses amis

Les fonctions dans cette catégorie sont `scanf()`, `fscanf()`, `sscanf()`, `vscanf()`, `vfscanf()`, `vsscanf()`, `wscanf()`, `fwscanf()`, `swscanf()`, `vwscanf()` et `vfwscanf()`.

Elles aussi prennent une chaîne de format en paramètre. Cette fois-ci la chaîne de format indique comment interpréter et convertir des données en entrée pour les placer dans des variables. Il faut bien sûr faire attention de passer des variables qui correspondent aux spécificateurs de conversion, mais il faut aussi faire attention à la taille des tampons qui servent à recevoir les chaînes de caractères lues. Puisqu'on ne connaît généralement pas la longueur exacte des chaînes de caractères avant de les lire, il vaut mieux toujours indiquer la taille du tampon à un spécificateur de conversion vers une chaîne de caractères. Par exemple, au lieu de spécifier «`%s`», il est préférable de spécifier «`%20s`» si le tampon est de 20 caractères.

Avec glibc, il est aussi possible d'utiliser le drapeau «a» pour demander d'allouer dynamiquement une chaîne de la bonne longueur. Il faut alors passer un pointeur pour recevoir un pointeur vers la chaîne de caractères comme dans l'exemple 2.10. Il faut cependant savoir que ce drapeau n'est pas défini par la norme C ISO/IEC 9899 :1999. Un programme qui l'utilise n'est donc pas portable.

```
char **p;
scanf("%as", p);
```

Exemple 2.10: Utilisation de `scanf()` avec allocation dynamique

2.3.5 gets()

La fonction `gets()` est à peu près impossible à utiliser correctement. Elle prend un seul paramètre : un pointeur vers un tampon. Elle lit une ligne de `stdin` et la place dans le tampon sans aucune vérification pour les débordements. Le problème est que `stdin` sert habituellement d'entrée au programme et qu'il n'est habituellement pas possible de connaître à l'avance la taille maximale d'une ligne d'entrée. L'exemple 2.11 montre tout de même un exemple d'utilisation correcte de la fonction `gets()`.

Il faut noter que l'exemple utilise des fonctions qu'on retrouve sur les systèmes de type Unix et qu'il ne peut pas être compilé sous Windows sans une bibliothèque d'émulation des appels système de Unix. Il faut aussi remarquer que GCC donne un avertissement lors de l'édition des liens pour l'utilisation de `gets()`. Il considère qu'utiliser cette fonction est dangereux et qu'elle ne devrait pas être utilisée. Il est très rare d'avoir l'assurance que les lignes lues en entrée respecteront les attentes. Pour cette raison il est recommandé de ne jamais utiliser `gets()` et d'utiliser plutôt une fonction de remplacement comme `fgets()` à laquelle on spécifie la taille du tampon.

2.3.6 realpath() et getwd()

Ces fonctions inscrivent chacune un chemin dans un tampon. `realpath()` inscrit la forme canonique d'un chemin qu'on lui fournit et `getwd()` inscrit le chemin du répertoire de travail courant. Elles ne sont pas très difficiles à utiliser correctement, il y a seulement une chose à faire attention. Le tampon qu'on leur passe doit avoir une taille d'au moins `PATH_MAX` caractères pour éviter les débordements.

2.3.7 Autres fonctions dangereuses

Les fonctions `getopt()` et `getpass()` sont à surveiller car dans certaines implémentations de la bibliothèque C, elles peuvent être victime d'un débordement de tampon interne. C'est donc plus un problème d'implémentation que d'interface. Glibc n'a pas ces problèmes. En particulier, `getpass()` alloue autant de mémoire que nécessaire pour lire une ligne d'entrée.

Les fonctions `streadd()`, `strecpy()` et `strtrns()` sont aussi à surveiller, mais elles n'existent pas dans glibc et ne peuvent donc pas être mal utilisées.

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

int main()
{
        int fd[2];
        close(0);
        close(1);
        pipe(fd);

        if (!fork()) {
                printf("Bonjour\n");
        } else {
                char tampon[10];
                /* Si cette utilisation de gets est correcte
                   c'est parce qu'on sait d'où vient l'entrée. */
                gets(tampon);
                fprintf(stderr, "J'ai lu: %s\n", tampon);
        }

        return 0;
}
```

L'exécution donne :

```
$ ./gets
J'ai lu: Bonjour
```

Exemple 2.11: Une très rare utilisation correcte de `gets()`. L'utilisation de `gets()` est sécuritaire seulement parce que `stdin` est déconnecté de l'entrée contrôlée par l'utilisateur (`close()`) et reconnecté au programme de façon interne (`pipe()`).

Nous avons vu dans cette section que plusieurs des fonctions normalisées de la bibliothèque C sont difficiles à utiliser correctement de façon à éviter les débordements de tampons. Il existe souvent des fonctions de rechange qui permettent plus facilement de respecter les bornes, mais les programmeurs peuvent être découragés de les utiliser par le fait qu'elles n'offrent pas des interfaces cohérentes.

2.4 Non-respect d'une interface

Il y a non-respect de l'interface lorsqu'on passe à une fonction un ou des paramètres qui ne respectent pas les spécifications. La ligne peut parfois être floue entre cette section et la précédente. On pourrait affirmer que les risques de débordement présentés à la section précédente étaient tous dus à un non-respect d'interface. Contrairement aux interfaces sans vérification, nous traitons ici des cas où l'interface est pensée pour empêcher les débordements, mais où elle n'est pas utilisée correctement.

Par exemple, `strlen()` prend en paramètre un pointeur vers une chaîne de caractères terminée par un caractère nul. Lui passer une chaîne qui n'est pas terminée par un caractère nul est donc une erreur qui mène le plus souvent à un débordement de tampon.

Dans l'exemple 2.12, le tampon est trop court pour contenir toute la chaîne. Il n'y a pas de

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

int main()
{
        char *chaine = "Bonjour à tous!";
        char tampon[8];
        strncpy(tampon, chaine, sizeof(tampon));
        printf("%d\n", strlen(tampon));
        return 0;
}
```

L'exécution de ce programme donne :

```
$ ./strlen
21
```

Exemple 2.12: Non-respect de l'interface de `strlen()`

débordement de tampon lors de la copie puisque la fonction `strncpy()` permet de faire respecter les bornes du tampon. Ceci a toutefois eu pour conséquence que le caractère nul pour indiquer la fin de la chaîne ne peut pas être copié. Le `strlen()` effectue donc un débordement de tampon et donne un résultat différent de celui attendu. En effet, il y a clairement accès en dehors des bornes puisque 21 caractères sont comptés alors que la chaîne copiée n'en compte que 8.

2.5 Écart d'un

Un *écart d'un*[6] est une erreur dans laquelle le résultat des calculs a une unité de différence avec la réponse attendue. Ceci se produit généralement lorsque le programmeur fait une erreur dans une formule ou dans la condition d'une boucle, soit parce qu'il oublie un item, souvent le caractère nul, qu'il en compte un de trop, ou qu'il se trompe de base pour les indices (1 vs 0). Les cas qui nous intéressent sont ceux où l'écart d'un donne lieu à un débordement de tampon. L'erreur se produit habituellement lors de la manipulation d'indices.

L'exemple 2.13 montre un cas typique d'écart d'un. Dans cet exemple, l'erreur est d'utiliser

```
#include <ctype.h>

void ecart()
{
        unsigned char tampon[8];
        unsigned char chaine[8] = "bonjour";
        int i;
        /* La boucle transforme chaine en majuscules */
        for (i=0; i<=8; i++)
                tampon[i] = toupper(chaine[i]);
}

int main()
{
        ecart();
        return 0;
}
```

Exemple 2.13: Écart d'un

l'opérateur <= plutôt que l'opérateur <. Regardons le résultat de l'exécution.

Exécution de l'exemple 2.13

```
$ ./ecart1
Erreur de segmentation
```

Un simple écart d'un est donc suffisant pour faire planter un programme. Nous verrons d'ailleurs au chapitre 3 qu'il peut être la cause d'une vulnérabilité très sérieuse.

Il faut remarquer que ce ne sont pas tous les écarts d'un qui nous intéressent. Par exemple, il est clair qu'un écart d'un qui aurait pour conséquence d'ignorer à tout coup le dernier caractère d'un tampon ne peut pas faire déborder ce tampon. Bien qu'une telle erreur puisse poser d'autres problèmes dans le programme, et même des problèmes de sécurité, ce cas est en dehors de la portée de ce rapport.

[6] Nous utilisons «écart d'un» comme traduction de l'expression anglaise *off by one error*.

2.6 Manipulation d'une chaîne de format

Lorsqu'on permet à un utilisateur de saisir ou de manipuler une chaîne de caractères qui est par la suite utilisée comme chaîne de format, on s'expose à certains problèmes de sécurité. En effet, les fonctions qui demandent une chaîne de format utilisent un nombre variable d'arguments et c'est le contenu de la chaîne de format qui définit le nombre et le type des arguments utilisés par la fonction.

Dans la bibliothèque C, les fonctions les plus susceptibles de causer des problèmes sont :

`printf()`, `fprintf()`, `sprintf()`, `snprintf()`, `vprintf()`, `vfprintf()`, `vsprintf()`, `vsnprintf()`, `wprintf()`, `fwprintf()`, `swprintf()`, `vwprintf()`, `vfwprintf()`, `vswprintf()`, `syslog()` et `vsyslog()`.

Les deux dernières ne sont pas normalisées, mais elles existent, elles sont utilisées fréquemment sous `Linux` et elles se comportent comme les fonctions de la famille de `printf()`. Les fonctions de la famille de `scanf()` peuvent aussi théoriquement causer des problèmes, mais il est plus difficile d'imaginer un programmeur passer par erreur autre chose qu'une chaîne de format à ces fonctions. Il faut aussi garder à l'esprit qu'un programmeur peut définir des fonctions qui prennent en paramètres une chaîne de format et qui appellent `vfprintf()`, `vsprintf()`, `vsyslog()` ou une autre fonction du genre à l'interne. De telles fonctions peuvent aussi causer les mêmes problèmes.

Les langages C et C++ ne spécifient aucun mécanisme pour s'assurer que le nombre et le type des arguments passés à une fonction et utilisés par celle-ci sont les mêmes lorsque la fonction est déclarée avec un nombre variable d'arguments. Il est donc possible que les arguments spécifiés dans la chaîne de format ne correspondent pas à des arguments effectivement passés par l'appelant de la fonction. Sachant ceci, il est facile d'imaginer qu'un utilisateur avec des intentions malveillantes puisse obtenir des informations auxquelles il n'a pas droit.

Dans l'exemple 2.14, *secret* représente des données qui sont manipulées par le programme, mais qui ne devraient jamais être affichées à l'utilisateur. Regardons maintenant le résultat d'une exécution avec pour objectif d'obtenir des informations secrètes.

Exécution de l'exemple 2.14

```
$ ./format1
Entrez votre nom: %X%X%X%X%X%X%X%X %s
Bonjour 644013608080496FC8049728BFFFFBE84004CF1840138E48400097C0 Code: 007
```

Dans cette chaîne de format, chaque «`%X`» demande l'affichage en format hexadécimal du «paramètre» suivant. Puisque aucun paramètre supplémentaire n'a été passé à la fonction `printf()`, chaque «`%X`» correspond à la valeur suivante qui se trouve sur la pile. La pile contient entre autres les variables locales et les paramètres des fonctions appelées. Un «`%s`» interprète une valeur sur la pile comme un pointeur vers une chaîne de caractères et en affiche le contenu. On voit tout de suite qu'une mauvaise utilisation des chaînes de format peut donner lieu à des problèmes de sécurité, ou plus précisément, de confidentialité. Nous verrons au chapitre 4 que le problème ne se limite pas à la confidentialité.

```
#include <stdio.h>

int main()
{
        char *secret = "Code: 007";
        static char entree[100] = {0};
        printf("Entrez votre nom: ");
        fgets(entree, sizeof(entree), stdin);
        printf("Bonjour ");
        printf(entree);
        return 0;
}
```

Voici un exemple d'exécution normale :

```
$ ./format1
Entrez votre nom: Patrice
Bonjour Patrice
```

Exemple 2.14: Manipulation de chaîne de format

L'erreur dans l'exemple précédent consiste à fournir à `printf()` une chaîne de caractères arbitraire au lieu d'une chaîne de format. Au lieu de

```
printf(entree)
```

on aurait dû utiliser

```
printf("%s", entree)
```

On aurait aussi pu combiner les deux `printf()` de la façon suivante :

```
printf("Bonjour %s", entree)
```

L'important est de s'assurer de ne pas passer une chaîne de caractères arbitraire comme chaîne de format.

Il faut noter que les vulnérabilités de chaîne de format sont présentes dans des programmes réels et depuis longtemps. Cependant, elles ne sont reconnues publiquement comme une classe importante de vulnérabilités que depuis quelques années. C'est en septembre 1999 que les premières discussions publiques ont eu lieu à leur sujet alors qu'on discutait du problème dans ProFTPD, un serveur FTP [Twi99b, Twi99a]. Mais ce n'est qu'en juin 2000 qu'on a vraiment commencé à considérer les chaînes de format comme une classe de vulnérabilité importante lorsque plusieurs attaques ont été publiées pour divers logiciels [CBB+01].

Il faut aussi rappeler que les vulnérabilités de chaîne de format ne sont pas des débordements de tampons. Il est toutefois possible de faire un parallèle entre les deux types de problème. Dans les deux cas, si on simplifie un peu et qu'on considère les paramètres variables comme un tableau de paramètres, on veut éviter d'accéder aux éléments à l'extérieur des bornes du tableau.

2.7 Réponses par défaut à des messages

Dans Windows, il est possible d'envoyer un message à n'importe quel programme qui a une fenêtre sur le bureau, même si cette fenêtre est cachée et que le programme s'exécute sous le compte d'un autre utilisateur qui a plus de privilège. Par exemple, les anti-virus sont souvent des programmes privilégiés avec des fenêtres cachées, et il est possible de leur envoyer des messages à partir de n'importe quel compte.

Windows permet de gérer la taille des entrées acceptées par une boîte de saisie de texte. On peut entre autres envoyer un message à une fenêtre et lui demander de modifier la taille des entrées acceptées. Par défaut, si le programme n'a pas été pensé pour spécifiquement éviter ce comportement, cette taille est changée, même si l'expéditeur est un programme qui s'exécute sans privilèges spéciaux. Lorsqu'un programme se fie uniquement à Windows pour contrôler la taille de certaines chaînes de caractères, il peut avoir des mauvaises surprises, comme des débordements de tampons.

Dans [Foo02a, Foo02b], on explique en détail ce problème et plusieurs autres problèmes reliés aux réponses par défaut aux messages provenant de sources non fiables. On y apprend que la plupart des programmes Windows permettent à un utilisateur de faire exécuter du code arbitraire sans qu'il n'y ait de débordement de tampon, ni d'écrasement d'adresse de retour, ni d'écrasement d'aucun pointeur. En effet, le message `WM_TIMER` permet de passer l'adresse d'une fonction à exécuter. Par défaut, cette fonction est exécutée sans aucune vérification de sécurité. Le plus beau dans tout ça c'est que ce n'est pas un défaut de Windows [Mic02].

Chapitre 3

Conséquences des débordements de tampons

Dans ce chapitre, nous verrons de façon générale quelles peuvent être les conséquences d'un débordement de tampon, et de façon plus spécifique les techniques qui permettent de les exploiter.

3.1 Exécution correcte

La présence possible ou réelle d'un débordement de tampon ou d'une manipulation de chaîne de format n'entraîne pas automatiquement un comportement anormal d'un programme. Par exemple, les situations suivantes ne causent habituellement pas de problème lorsqu'il y a des possibilités de débordement de tampon :

- les entrées fournies pas l'utilisateur respectent la taille du tampon même si le programme ne la vérifie pas ;
- la mémoire qui suit un tampon n'est pas encore allouée ou initialisée et elle ne le sera pas avant d'avoir terminé d'utiliser le tampon ;
- la mémoire qui suit le tampon a déjà été utilisée mais ne l'est plus ;
- la mémoire écrasée par le débordement de tampon est remplacée par la même valeur ou une valeur équivalente à ce qui s'y trouvait déjà pour la logique du programme.

De même, pour les manipulations de chaîne de format, il est très probable que la chaîne lue en entrée ne contienne pas de spécificateur de conversion. Dans ces cas aussi l'exécution du programme est habituellement correcte.

Les cas précédents sont très généraux et on pourrait trouver plusieurs exemples où ils se produisent. Cependant, nous n'explorerons pas plus à fond ces situations car nous cherchons plutôt à montrer les conséquences observables d'un débordement de tampon, ou à tout le moins, celles qui risquent d'avoir des impacts sur la sécurité.

3.2 Exception

Certains langages génèrent une exception lorsqu'on tente d'accéder à un élément à l'extérieur des bornes d'un tableau. On pourrait croire qu'avec ces langages on est immunisé contre les problèmes liés aux débordements de tampons, mais ce serait à tort. Bien qu'il n'y ait pas de débordement lors d'une tentative d'accès à des éléments se trouvant à l'extérieur des bornes des tableaux, une exception est tout de même générée et elle peut avoir des conséquences. Nous étudierons ces conséquences en fonction du type de gestion des exceptions, à savoir une gestion correcte, une absence de gestion et une gestion incorrecte.

Les exemples qui suivent sont en Java puisque le C et le C++ ne génèrent pas d'exceptions lors d'un accès hors bornes.

3.2.1 Gestion correcte des exceptions

Lors d'un accès susceptible d'être en dehors des bornes d'un tableau, une possibilité souvent utilisée dans les langages qui génèrent une exception consiste à ignorer complètement les contraintes de taille du tableau dans le coeur du traitement. On se fie alors que le moteur d'exécution détectera un accès à l'extérieur des bornes du tableau et qu'il le signalera au programme au moyen d'une exception.

L'exemple 3.1 montre un programme qui utilise cette technique. Ce programme initialise les 10

```
class Debordement2
{
        public static void main(String arg[])
        {
                int tab[] = new int[10];
                int i = 0;
                try {
                        while (true) {
                                tab[i] = i;
                                i++;
                        }
                } catch (ArrayIndexOutOfBoundsException e) {
                        System.out.println("Fin du tableau");
                }
        }
}
```

Exemple 3.1: Gestion correcte des exceptions

éléments du tableau, puis quand vient le temps d'initialiser le 11^{e} élément, le moteur d'exécution lance une exception qui est attrapée par le programme. Ce programme ne pose aucun problème parce que même s'il ne vérifie pas les bornes du tableau dans la condition de la boucle, ça ne l'empêche pas de gérer correctement les débordements via les exceptions.

Nous ne nous attarderons pas plus à la gestion correcte des exceptions car, encore une fois, nous sommes plus intéressés par les cas qui sont incorrects.

3.2.2 Absence de gestion des exceptions

Avec le langage Java, lorsqu'on tente d'accéder à un élément à l'extérieur des bornes d'un tableau et qu'on ne gère aucunement les exceptions, le fil d'exécution s'arrête et un message est affiché pour indiquer ce qui s'est passé. S'il est difficile d'imaginer comment un attaquant pourrait utiliser ce comportement pour violer la confidentialité ou l'intégrité d'un système, il est beaucoup plus facile de voir que la disponibilité d'un système pourrait être affectée. En effet, un attaquant pourrait utiliser un débordement de tableau non géré pour monter un déni de service.

On ne peut toutefois pas affirmer que les problèmes d'intégrité et de confidentialité sont impossibles lorsqu'un programme omet de gérer les exceptions, en particulier celles qui indiquent le débordement d'un tableau. Ces problèmes pourraient se manifester dans des systèmes plus complexes où plusieurs programmes ou «fils d'exécution» interagissent. Si un d'entre eux ne complétait pas ce qu'il devait faire avant de terminer son exécution, un autre pourrait alors poursuivre son exécution dans un état qui ferait en sorte que la confidentialité ou l'intégrité serait compromise.

3.2.3 Gestion incorrecte des exceptions

Il faut être très prudent lorsqu'on gère les exceptions parce qu'il est possible de se retrouver en train de gérer une exception qu'on n'avait pas prévue. Lorsqu'une exception n'est pas gérée, elle fait normalement arrêter le programme, ce qui permet de voir qu'il y a un problème et de le corriger. Par contre, lorsqu'une exception est gérée par erreur, les conséquences sont habituellement moins apparentes et on risque de ne pas s'en apercevoir immédiatement.

Deux sortes d'erreurs peuvent faire en sorte qu'une exception est gérée alors qu'elle ne devrait pas l'être. Il y a d'abord les erreurs sur les types d'exception. Considérons le programme de l'exemple 3.2. Ce programme attend deux paramètres, le dividende et le diviseur, et affiche le quotient des deux nombres pour une division entière. Il gère une exception dans le but de détecter les divisions par zéro et d'afficher un message significatif dans cette situation.

Nous observons qu'il se comporte comme prévu avec des paramètres attendus. Mais qu'en est-il si on omet de lui passer des paramètres ?

Exécution de l'exemple 3.2

```
$ java Diviser
Division par 0
```

Ici le programme a d'abord tenté d'accéder au premier élément du tableau des paramètres qui n'en comptait aucun. Il a donc généré une exception de type `ArrayIndexOutOfBoundsException`. Il faut cependant savoir que cette classe d'exception hérite de `RuntimeException`. L'exception a donc été attrapée et interprétée par le programme comme une division par zéro, alors qu'il n'y a même pas eu de division.

```
class Diviser
{
        public static void main(String arg[])
        {
                try {
                        System.out.println(
                                (float)Integer.parseInt(arg[0]) /
                                Integer.parseInt(arg[1]));
                } catch (RuntimeException e) {
                        System.out.println("Division par 0");
                }
        }
}
```

Voici des exemples d'exécution normale :

```
$ java Diviser 6 2
3

$ java Diviser -15 3
-5

$ java Diviser 4 0
Division par 0
```

Exemple 3.2: Gestion des exceptions avec un type trop générique

Pourtant, sans gestion d'exception, un message d'erreur indiquerait explicitement qu'il y a eu une erreur d'accès en dehors des bornes d'un tableau en spécifiant que l'exception est de type `ArrayIndexOutOfBoundsException`. Il faut donc retenir qu'une exception quelconque, mais en particulier une exception d'accès à un élément à l'extérieur des bornes d'un tableau puisque c'est ce cas qui nous intéresse ici, peut faire transférer l'exécution d'un programme directement à du code prévu pour gérer un autre type d'erreur. Le programme continue ensuite de s'exécuter normalement, possiblement sans avoir complété ce qu'il devait faire. On peut imaginer que si ce problème survenait dans une partie sensible d'un programme, comme la partie qui gère l'authentification ou la gestion des accès, les conséquences sur la sécurité peuvent être importantes.

Dans l'exemple précédent, la solution au problème serait de gérer les `ArithmeticException` qui sont plus spécifiques que les `RuntimeException`. En gérant toujours les exceptions du type le plus spécifique possible, on évite les problèmes liés à la gestion erronée d'un type d'exception non attendu. Ça ne règle cependant pas tous les problèmes.

Une autre sorte d'erreur peut avoir des conséquences similaires. Il s'agit de l'utilisation d'une gestion d'exceptions avec une portée trop grande. Lorsqu'une exception est lancée, elle est attrapée par le «premier `catch`» qui correspond à la classe de l'exception. Ce `catch` peut cependant être relativement loin de la véritable erreur car un bloc `try` peut couvrir plusieurs lignes de code et une méthode qui fait beaucoup de traitement peut être appelée.

Considérons l'exemple 3.3. La gestion des exceptions n'exécute aucun code, elle marque simple-

```
.

class Debordement3
{
        public static void main(String arg[])
        {
                int tab[] = new int[10];
                try {
                        // Initialisation
                        int v = Integer.parseInt(arg[0]);
                        int i = 0;
                        while (true) {
                                tab[i] = v;
                                i++;
                        }
                } catch (ArrayIndexOutOfBoundsException e) {
                        // Fin de l'initialisation
                }
                // Le traitement sur les données peut commencer
        }
}
```

Exemple 3.3: Gestion des exceptions avec une portée trop grande

ment la fin de l'initialisation du tableau et le début du traitement. Le problème est que l'exception `ArrayIndexOutOfBoundsException` peut provenir autant d'`arg[0]` que de `tab[i]` même si le code

ne semble considérer que ce dernier cas. Si l'exception vient d'`arg[0]`, le programme pourrait commencer le traitement en utilisant un tableau qui n'est pas initialisé. Selon ce que le tableau représente et le traitement qui est effectué par la suite sur ce tableau, il pourrait y avoir des impacts sur la sécurité.

Une solution pour régler le problème ici serait de diminuer la portée du bloc `try` pour ne couvrir que le bloc `while`. De cette façon on aurait l'assurance que l'exception vient toujours du `tab[i]`.

La portée excessive d'un `try` peut se manifester même lorsque le bloc `try` ne contient qu'une seule ligne de code. Par exemple, si cette ligne est un appel de méthode, cette méthode peut générer un même type d'exception pour des situations différentes. En général, pour éviter ces situations, on cherche à gérer les exceptions plus près de l'endroit où elles se produisent, c'est-à-dire directement au niveau de la méthode où la condition d'erreur se produit. Si on a besoin de lancer une exception à l'appelant, il vaut mieux que ce soit une exception spécifique pour chaque situation différente.

Nous n'avons pas trouvé de cas documenté où la gestion incorrecte des exceptions donnait lieu à une vulnérabilité causée par un débordement de tampon. Toutefois, cette absence de cas documenté ne signifie pas que le problème n'existe pas, et les exemples précédents nous convainquent qu'il vaut mieux ne pas ignorer cette possibilité.

Il faut donc retenir que dans un langage qui effectue la vérification des bornes à l'exécution, même si techniquement il ne peut pas y avoir de débordement de tampon, les «tentatives de débordement» peuvent tout de même avoir des conséquences importantes. Il est donc justifié, même pour ces langages, de chercher à éliminer les possibilités de débordement de tampon avant l'exécution. Les techniques d'analyse statique qui seront présentées au chapitre 5 pourraient donc être appliquées à ces langages.

3.3 Fin anormale

Une fin anormale se produit lorsqu'un programme effectue quelque chose de grave et non contrôlé. Les débordements de tampons peuvent provoquer une fin anormale de plusieurs façons différentes. La première se produit lorsqu'un débordement modifie l'adresse de retour d'une fonction vers une adresse invalide. Lors du retour de la fonction, l'adresse ne correspond pas à de la mémoire pouvant être exécutée et il se produit ce qu'on appelle une erreur de segmentation. C'est ce qui se passe dans l'exemple 3.4.

Une variante de l'erreur de segmentation se produit lorsqu'un débordement écrase un pointeur qui est par la suite utilisé. Il se peut que le pointeur ainsi écrasé pointe vers une adresse invalide. De la même façon que pour un retour, il y a une erreur de segmentation parce que la mémoire est inaccessible en lecture ou en écriture, selon le cas. C'est ce qui se passe dans l'exemple 3.5.

Un autre type d'erreur peut provoquer une fin anormale. Il s'agit de l'exécution d'instructions illégales. Sur un processeur, ce ne sont pas toutes les séquences d'octets qui correspondent à des instructions bien définies. Celles qui ne le sont pas sont appelées instructions illégales. Il faut noter que les processeurs de la famille IA-32 possèdent une instruction `UD2` définie explicitement comme illégale. Elle permet par exemple de vérifier la détection des instructions illégales par les concepteurs d'un système d'exploitation. Utiliser cette instruction plutôt que n'importe quelle autre donne

```
int main()
{
        short *tab[10];
        int i;
        for (i=0; i<30; i++)
                // Adresse invalide
                tab[i] = 0;
        // Retour à une adresse invalide
        return 0;
}
```

Voici l'exécution :

```
$ ./faute1
Erreur de segmentation
```

Exemple 3.4: Fin anormale provoquée par un retour à une adresse invalide. L'adresse invalide ici est celle du pointeur NULL (0).

```
int main()
{
        short *tab[10];
        int i;
        for (i=0; i<30; i++)
                // Adresse invalide
                tab[i] = 0;
        // Utilisation d'une adresse invalide
        *tab[0] = 1;
        return 0;
}
```

Voici l'exécution :

```
$ ./faute2
Erreur de segmentation
```

Exemple 3.5: Fin anormale provoquée par l'utilisation d'un pointeur invalide. L'adresse invalide ici est celle du pointeur NULL (0).

l'assurance qu'il s'agit bien d'une instruction illégale car les autres instructions illégales pourraient être définies dans des générations suivantes des processeurs de la famille IA-32. Certaines instructions peuvent aussi être considérées illégales parce qu'elles sont réservées au système d'exploitation. L'exemple 3.6 exécute une instruction illégale.

```
int main()
{
        short *tab[10];
        // Instruction UD2 sur processeur intel (invalide)
        static short ud2 = 0x0b0f;
        int i;
        for (i=0; i<30; i++)
                tab[i] = &ud2;
        return 0;
}
```

Voici l'exécution :

```
$ ./illegal
Instruction illégale
```

Exemple 3.6: Fin anormale provoquée une instruction illégale. L'instruction `UD2` permet de générer à coup sûr une exception d'instruction illégale.

Tous ces comportements font en sorte qu'un pirate peut utiliser un débordement de tampon pour provoquer des dénis de service.

3.4 Exécution anormale

Si les débordements de tampons peuvent dans certains cas n'avoir aucun impact sur l'exécution d'un programme et si dans d'autres cas ils peuvent provoquer la fin anormale d'un programme, il existe aussi toute une gamme de conséquences possibles entre ces deux extrêmes.

Il suffit qu'une variable suive en mémoire un tampon qui peut déborder pour que son contenu puisse être modifié par un débordement. Selon ce que représente cette variable, le résultat de la modification peut être plus ou moins grave. Par exemple, il n'est pas difficile de se convaincre que l'écrasement d'un nom de fichier, de l'identifiant d'un utilisateur ou de la description de ses autorisations peut provoquer des conséquences importantes dans un programme privilégié.

D'autres variables peuvent avoir des effets indirects, par exemple, une variable qui indique le nombre d'essais restant à un utilisateur pour s'authentifier avant que son compte ne soit désactivé. Il serait possible d'utiliser une telle variable pour obtenir un nombre illimité d'essais avant de trouver le bon mot de passe.

Les possibilités d'attaque varient grandement en fonction des variables qui peuvent être écrasées par un débordement. Le reste du chapitre montre certains principes qui permettent à un attaquant de contrôler l'exécution anormale d'un programme.

3.5 Exécution de code arbitraire

Parmi les conséquences possibles des débordements de tampons, celle qui est la plus grave est probablement l'exécution de code arbitraire. Cette conséquence va plus loin que l'exécution anormale puisque la possibilité d'exécuter du code arbitraire simplifie grandement la tâche d'une personne qui veut exploiter un débordement de tampon. Il n'est pas évident à première vue comment un débordement de tampon, dans une zone qui est normalement destinée à recevoir des données, peut permettre l'exécution de code arbitraire. Nous verrons que plusieurs techniques peuvent être utilisées pour permettre ceci.

3.5.1 Écraser l'adresse de retour

La plupart des langages qui permettent la récursivité utilisent une pile pour gérer les paramètres, les adresses de retour et les variables locales des procédures. C'est le cas notamment des langages C et C++.

Lors de l'appel à une fonction, l'appelant empile d'abord les paramètres de cette fonction, puis il empile l'adresse de retour avant de transférer le contrôle à la fonction. L'adresse de retour est normalement celle de l'instruction qui suit l'appel de la fonction. Au début de l'exécution de la fonction appelée, cette dernière réserve de l'espace sur la pile pour ses variables locales. Elle utilise aussi cet espace pour sauvegarder la valeur de certains registres importants qui sont modifiés dans la fonction mais qui doivent revenir à leur valeur originale lors du retour à la fonction appelante. La figure 3.1 illustre l'organisation de la pile.

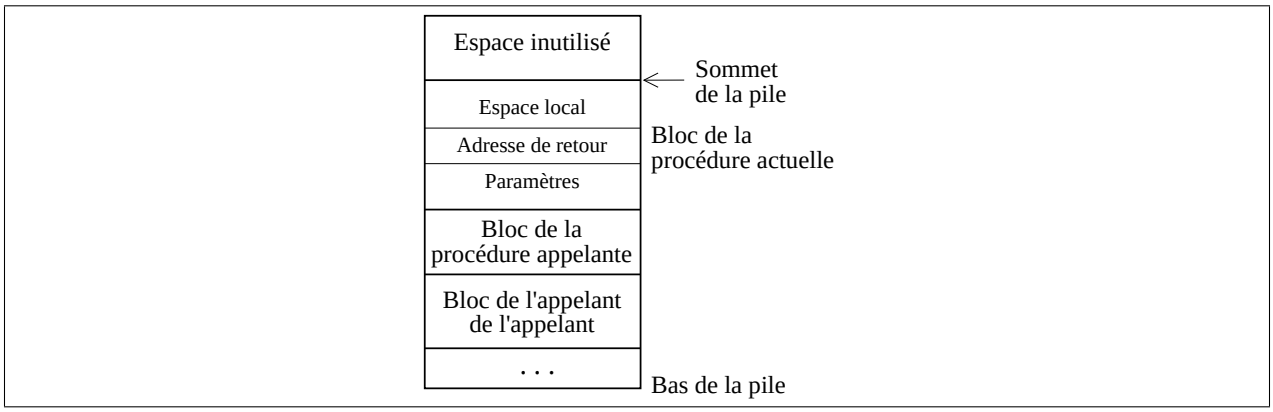

Figure 3.1: Pile d'exécution d'un programme. Ici, ce qu'on appelle le bloc (de pile) d'une procédure est composé des paramètres, de l'adresse de retour et de l'espace local de cette procédure.

Si on pouvait modifier à son gré l'adresse de retour qui se trouve sur la pile, on serait en mesure de transférer le cours d'exécution du programme vers un endroit arbitraire dès le moment où la procédure tente de retourner à l'appelant. Un débordement de tampon peut avoir comme conséquence d'écraser l'adresse de retour, mais certaines conditions doivent être réunies.

D'abord, le tampon doit se trouver sur la pile. Si le tampon n'était pas sur la pile, le débordement devrait couvrir tout l'espace d'adressage entre l'endroit où se trouve le tampon et la pile avant d'écraser l'adresse de retour. Il faut savoir qu'au démarrage d'un programme, par exemple sur l'architecture Linux/i386, cet espace d'adressage couvre approximativement 3 gigaoctets. Même

en ignorant le fait que le programme risquerait de manquer de mémoire avant de couvrir l'espace d'adressage requis, il reste que la plus grande partie de cet espace n'est pas allouée au programme et que le programme serait arrêté dès qu'il tenterait d'y accéder sur tout système d'exploitation qui offre la protection de l'espace d'adressage. C'est pourquoi on considère que le tampon doit se trouver sur la pile.

Ensuite, une adresse de retour doit se trouver en mémoire après le tampon. Bien qu'il soit techniquement possible d'effectuer un débordement de tampon du côté du début d'un tampon, il est très rare que ça se produise dans des programmes réels ; les débordements se produisent habituellement à la fin.

Avec ces connaissances, on s'empresse de retourner consulter la figure 3.1 pour s'apercevoir que la figure indique un bas et un sommet de pile, mais qu'elle n'indique pas dans quelle direction les adresses augmentent. La raison est simple, ça dépend de l'architecture utilisée, notamment du système d'exploitation et du processeur. Une vérification rapide dans le code source de glibc, la bibliothèque C utilisée dans la plupart des distributions Linux, montre que sur Linux, la pile grandit vers les adresses basses dans presque tous les cas. La seule exception concerne les processeurs PA-RISC de HP, où la pile grandit vers les adresses hautes. Il faut donc savoir que la pile peut grandir dans une direction ou dans l'autre selon le système d'exploitation et le processeur utilisé, mais retenons que le plus souvent elle grandit vers les adresses basses.

Imaginons un tampon qui se trouve dans l'espace local d'une procédure. Si ce tampon peut déborder, c'est d'abord le reste de l'espace local qui sera écrasé, c'est-à-dire la partie qui suit le tampon, et ensuite viendra le tour de l'adresse de retour. Les pirates cherchent donc à écraser l'adresse de retour de la procédure pour que leur code soit exécuté au moment où la procédure tente de retourner à son appelant. Pour ce faire, ils doivent résoudre trois problèmes.

Premièrement, un pirate doit réussir à placer son code[1] dans la mémoire du programme attaqué. Ceci peut se faire de plusieurs façons. À peu près toutes les sources d'entrée du programme sont susceptibles d'être utilisées par un pirate pour insérer son code dans le programme. Par exemple, il peut utiliser le flux d'entrée standard, un fichier, une communication réseau ou une variable d'environnement. Selon la méthode utilisée par le programme pour manipuler l'entrée en question, il est possible que le pirate n'ait pas une liberté absolue sur le code qu'il est en mesure de placer dans la mémoire du programme attaqué. Par exemple, si c'est en tant que chaîne de caractères que le code est lu par le programme, il n'est habituellement pas possible de faire passer des octets de valeur 0 au milieu de la chaîne puisque les caractères nuls sont utilisés pour indiquer la fin d'une chaîne de caractères. Dans ce cas, il doit écrire son code machine de manière à ce qu'il ne contienne pas d'octets ayant pour valeur 0. Il n'est généralement pas très difficile de modifier du code machine pour éviter les octets nuls.

Deuxièmement, il doit savoir à quel endroit en mémoire se trouve son code pour écraser l'adresse de retour avec la bonne valeur. Ce code peut se trouver aussi bien dans les données statiques du programme, dans la mémoire allouée dynamiquement ou sur la pile ; ça n'a pas d'importance. L'important est de connaître l'adresse. Si le pirate a une copie du programme qu'il peut exécuter dans un débogueur, il lui est plus facile de trouver cette information. S'il n'a pas accès à l'exécutable mais qu'il a le code source, il peut toujours le compiler et le déboguer pour obtenir une bonne idée

[1] On appelle souvent ce code *shellcode* parce qu'il exécute le plus souvent un interpréteur de commandes (*shell*) avec les privilèges d'un administrateur. On utilise aussi le terme *egg* qui fait référence à ce qu'on place dans le tampon (code, pointeur ou autres) qui déborde et qui permet à un attaquant d'obtenir ce qu'il veut.

de l'adresse. S'il n'a ni accès au code exécutable, ni au code source, il doit procéder par essais et erreurs.

Il faut noter que plusieurs facteurs peuvent faire en sorte que l'adresse exacte est difficile à prévoir pour le pirate. Par exemple, un code source compilé avec un compilateur différent ou des options de compilation différentes peut donner un exécutable différent. Aussi, lorsque la mémoire est allouée dynamiquement, il est possible que l'adresse soit différente d'une exécution à l'autre. Même l'adresse d'une zone de mémoire allouée sur la pile peut varier en fonction des appelants.

Quoi qu'il en soit, les pirates disposent d'un truc qui fait en sorte qu'ils n'ont pas besoin de connaître l'adresse exacte de leur code, mais seulement l'adresse approximative. En effet, un pirate n'a qu'à précéder son code d'une instruction qui n'a aucun effet, qu'on appelle `NOP`[2], et qui est répétée plusieurs fois. De cette façon, il est possible d'utiliser l'adresse de n'importe laquelle de ces instructions comme adresse de retour pour que le code du pirate soit exécuté.

Troisièmement, le pirate doit savoir où se situe l'adresse de retour par rapport au tampon qui déborde. La figure 3.2 montre de façon plus détaillée l'organisation de la pile. Le pirate qui veut écraser l'adresse de retour (*AR*) doit donc remplir le tampon (*Tampon*) puis le reste de l'espace

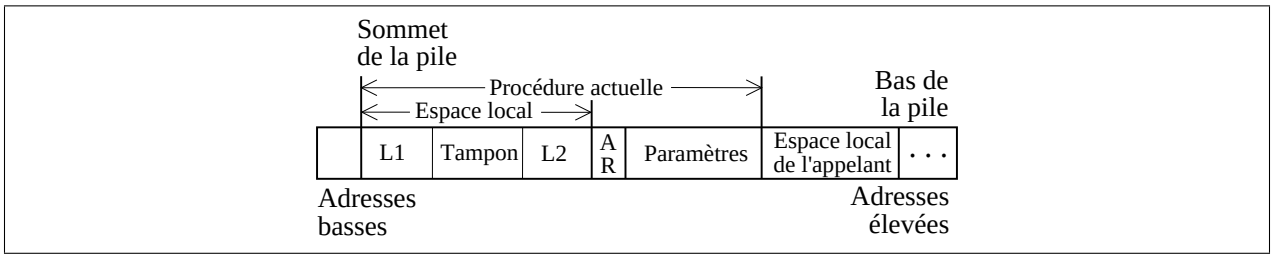

Figure 3.2: Pile qui grandit vers les adresses basses. Pour écraser l'adresse de retour (*AR*) il faut que *Tampon* déborde, d'abord sur *L2*, puis sur *AR*.

local, de la fin du tampon jusqu'à l'adresse de retour (*L2*). En théorie, il doit donc connaître la taille du tampon et celle du reste de l'espace local. Encore une fois, le pirate n'a pas à connaître la position exacte. Il peut commencer à répéter l'adresse où se trouve son code bien avant l'endroit exact où se trouve *AR* et il peut continuer à la répéter après *AR*. Il doit tout de même avoir une petite idée de la taille de *Tampon* et de *L2* car la pile n'est pas infinie, et s'il tente de dépasser le bas de la pile, le programme va planter avant que le contrôle soit transféré à son code, du moins sur un système d'exploitation qui offre la protection de la mémoire. D'un autre côté, s'il ne se rend pas à l'adresse de retour, son code ne sera pas exécuté non plus. Il dispose toutefois, entre ces deux extrêmes, d'une bonne marge de manoeuvre.

Il faut remarquer que rien n'empêche d'utiliser le tampon que l'on fait déborder pour y placer le code de l'attaque. Un seul débordement de tampon permet donc, du même coup, de placer le code dans l'espace d'adressage du programme et d'écraser l'adresse de retour pour qu'elle réfère à ce code. C'est d'ailleurs cette façon de faire que Aleph One décrit en détail dans [One96]. Il suggère de créer un tampon qui dépasse de 100 octets le tampon qu'on veut faire déborder. On construit ce tampon de façon à ce que le code à exécuter s'y situe au milieu et on remplit ce qui précède le code avec des instructions `NOP`, et ce qui le suit avec une adresse qu'on estime pointer dans les `NOP`. La figure 3.3 montre l'organisation de la pile une fois que le tampon a débordé de cette façon.

[2] `NOP` vient de l'anglais : *No Operation*.

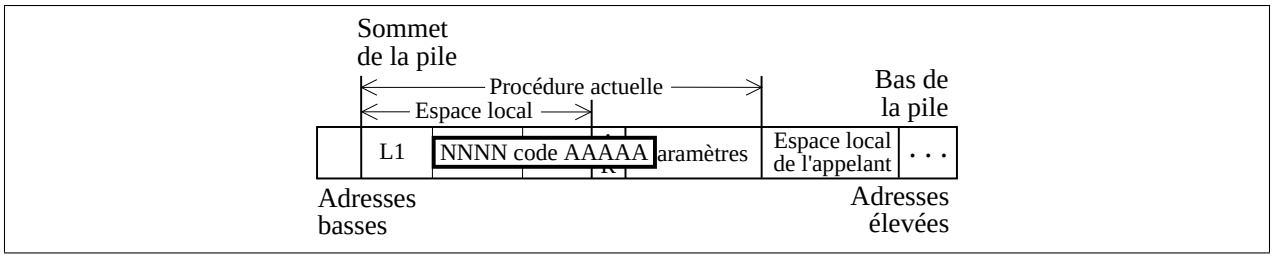

Figure 3.3: Pile avec un tampon qui a débordé. Les N représentent des instructions `NOP` et les A représentent une adresse qui se trouve à l'intérieur des `NOP`.

Cette façon de faire n'est cependant pas la seule. Si le tampon était trop petit et que l'adresse de retour était trop près du tampon, on pourrait placer les `NOP` et le code après l'adresse de retour et donc déborder plus loin dans les paramètres, voire dans l'espace de la procédure appelante, ou de l'appelant de cette dernière... Il faut simplement faire attention de ne pas dépasser la limite de la pile.

Lorsque la pile grandit vers les adresses hautes, on pourrait être porté à croire qu'un tampon qui déborde le fera vers les adresses élevées, où il n'y a pas de possibilité d'écraser une adresse de retour. Ce serait à tort. Bien que le débordement d'un tampon ne puisse pas écraser l'adresse de retour de la fonction dans laquelle il est déclaré, puisqu'il est placé plus loin en mémoire, ça n'empêche pas d'écraser l'adresse de retour d'une fonction appelée par celle où le tampon est déclaré. Examinons l'exemple 3.7.

```
#include <stdio.h>

int main()
{
        char tampon[10];
        /* On veut provoquer un
         * débordement de tampon */
        gets(tampon);

        return 0;
}
```

Exemple 3.7: Écrasement de l'adresse de retour d'une fonction appelée. Si la pile grandit vers les adresses hautes, c'est l'adresse de retour de `gets()` qui est écrasée. Si elle grandit vers les adresses basses, c'est l'adresse de retour de `main()` qui est écrasée.

La figure 3.4 montre l'organisation de la pile à l'entrée de la fonction `gets()` de l'exemple 3.7. On y remarque donc que si le tampon déborde, c'est l'adresse de retour de `gets()` qui est écrasée et que si on parvient à l'écraser avec la bonne valeur, au lieu de retourner dans `main()`, l'exécution pourrait par exemple se poursuivre dans le tampon. Pour que ça fonctionne, il faut tout de même faire attention de ne pas déborder plus loin que l'adresse de retour, car `gets()` conserve, dans son espace local, un pointeur vers l'endroit où elle est rendue à écrire dans le tampon. Avec `glibc`, il n'y a pas de problème à écraser le paramètre de `gets()` puisqu'elle prend une copie de ce paramètre à son entrée.

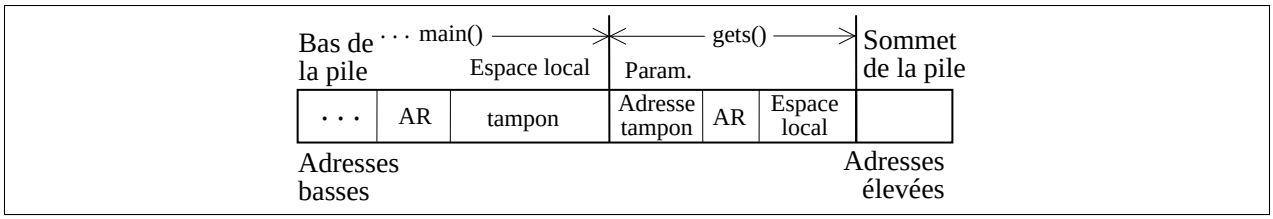

Figure 3.4: Organisation de la pile à l'entrée de la fonction `gets()` de l'exemple 3.7 lorsque la pile grandit vers le haut. Les *AR* représentent des adresses de retour différentes, une pour `gets()` et une autre pour `main()`. Attention, le sommet de la pile n'est plus du même côté !

La technique décrite dans cette section et ses variantes sont très utilisées par les pirates parce qu'elles ont l'avantage d'être relativement simples et elles s'appliquent à un grand nombre de programmes vulnérables. Les sections qui suivent donnent des alternatives que les pirates peuvent utiliser dans certaines situations où il n'est pas possible d'écraser l'adresse de retour, ou du moins pas directement.

3.5.2 Écraser le pointeur de bloc de pile sauvegardé

Cette technique permet d'exploiter un débordement de tampon et d'exécuter du code arbitraire dans certaines conditions particulières où il n'y a pas de possibilité de déborder par plus d'un octet (écart d'un). Elle est décrite en détail dans [klo99]. Pour comprendre le fonctionnement de cette technique, il faut d'abord connaître un peu mieux le fonctionnement du code généré par un compilateur pour une fonction C ou C++.

Ce qui suit est écrit pour les processeurs de la famille IA-32, mais s'applique également aux autres processeurs petit-boutistes[3] qui ont une pile grandissant vers le bas, et qui ont des mots de 32 bits.

Les compilateurs réservent en général un registre du processeur pour conserver l'adresse du bloc de pile [4]. Ce registre, appelé `EBP`, pointe généralement tout près de l'adresse de retour, de sorte que les paramètres de la fonction se trouvent à une adresse plus élevée que le pointeur de bloc de pile (déplacement positif) et les variables dans l'espace de travail local de la fonction se trouvent à des adresses plus basses (déplacement négatif). Cette façon de faire a l'avantage de permettre au compilateur d'utiliser un déplacement constant pour accéder aux paramètres et aux variables locales. Ça ne serait pas possible si c'était le pointeur de pile (`ESP`) qui était utilisé directement, puisque sa valeur peut changer au cours de l'exécution d'une fonction.

Ce pointeur doit donc être sauvegardé sur la pile à l'entrée d'une fonction avant d'être initialisé pour pointer le bloc de pile de cette fonction. À la fin d'une fonction, sa valeur doit être récupérée de manière à ce que la fonction appelante ait accès à son bloc de pile. L'exemple 3.8 montre comment ça se passe au niveau du processeur. Dans `fonct()`, `EBP` est d'abord sauvegardé sur la pile, puis `EBP` est initialisé pour pointer au sommet de la pile. Ensuite, `ESP` est décrémenté pour créer l'espace

[3]Un processeur petit-boutiste stocke l'octet de poids faible d'un mot à l'adresse la plus basse. À l'opposé, un processeur gros-boutiste stocke l'octet de poids fort à l'adresse la plus basse. Ces termes viennent de l'anglais *little-endian* et *big-endian* et font référence récit de Jonathan Swift, «Voyages de Gulliver».

[4]Un bloc de pile (*stack frame*) est associé à un appel de fonction et contient les paramètres, les données locales d'une fonction ainsi que l'adresse de retour.

Ici `main()` appelle `fonct(1, 2, 3)`.

```
0x8048431 <main+9>:     push   $0x3
0x8048433 <main+11>:    push   $0x2
0x8048435 <main+13>:    push   $0x1
0x8048437 <main+15>:    call   0x80483c0 <fonct>
0x804843c <main+20>:    add    $0x10,%esp
```

Ce qui suit est le prologue et l'épilogue de `fonct()`. Il faut savoir que `fonct()` déclare entre autres un tableau de 60 caractères dans son espace local.

```
0x80483c0 <fonct>:      push   %ebp
0x80483c1 <fonct+1>:    mov    %esp,%ebp
0x80483c3 <fonct+3>:    sub    $0x58,%esp
...
0x8048424 <fonct+100>:  leave
0x8048425 <fonct+101>:  ret
```

Il faut savoir que l'instruction `leave` est l'équivalent de :

```
mov    %ebp,%esp
pop    %ebp
```

Exemple 3.8: Code assembleur typique correspondant à l'appel et au retour d'une fonction C.

local. Quatre-vingt-huit octets (58_{16}) sont ainsi réservés. L'instruction `leave` permet de remettre les valeurs d'origine dans `ESP` et `EBP` avant de retourner à la fonction appelante.

La figure 3.5 donne le schéma de la pile après l'exécution de l'instruction `sub`. Sur la pile, les

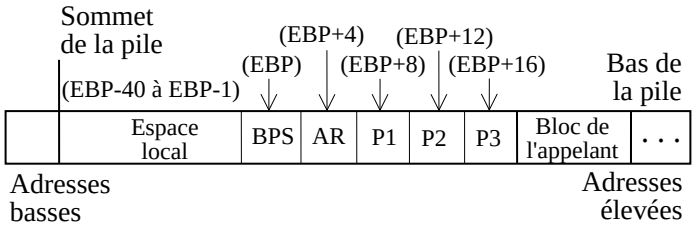

Figure 3.5: Organisation détaillée d'un bloc de pile. *P1* à *P3* représentent les trois paramètres de la fonction, *AR* est l'adresse de retour et *BPS* est le pointeur de bloc de pile sauvegardé, soit l'ancienne valeur de `EBP`.

paramètres occupent généralement 4 octets chacun. Dans l'espace local par contre, les variables peuvent être organisées de façon beaucoup plus souple et il est fréquent de retrouver des structures et des tableaux.

Maintenant nous allons voir comment on peut arriver à faire exécuter du code arbitraire avec un débordement d'un seul octet. Nous supposons que le tampon se situe sur la pile tout juste avant le pointeur de bloc de pile sauvegardé, *BPS*. On ne peut donc qu'écraser l'octet de poids faible. Nous appelons BPS_M la valeur de *BPS* avec l'octet de poids faible modifié. La figure 3.6 montre cette

situation.

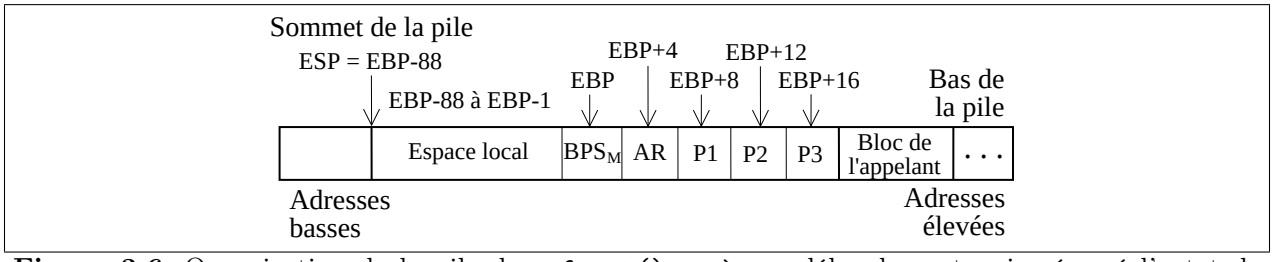

Figure 3.6: Organisation de la pile dans `fonct()` après un débordement qui a écrasé l'octet de poids faible du pointeur de bloc de pile.

À la fin de `fonct()`, l'instruction `leave` est exécutée et on se retrouve dans l'état décrit par la figure 3.7. On remarque que `EBP` a pris la valeur du pointeur de bloc de pile modifié. Si *BPS* n'avait

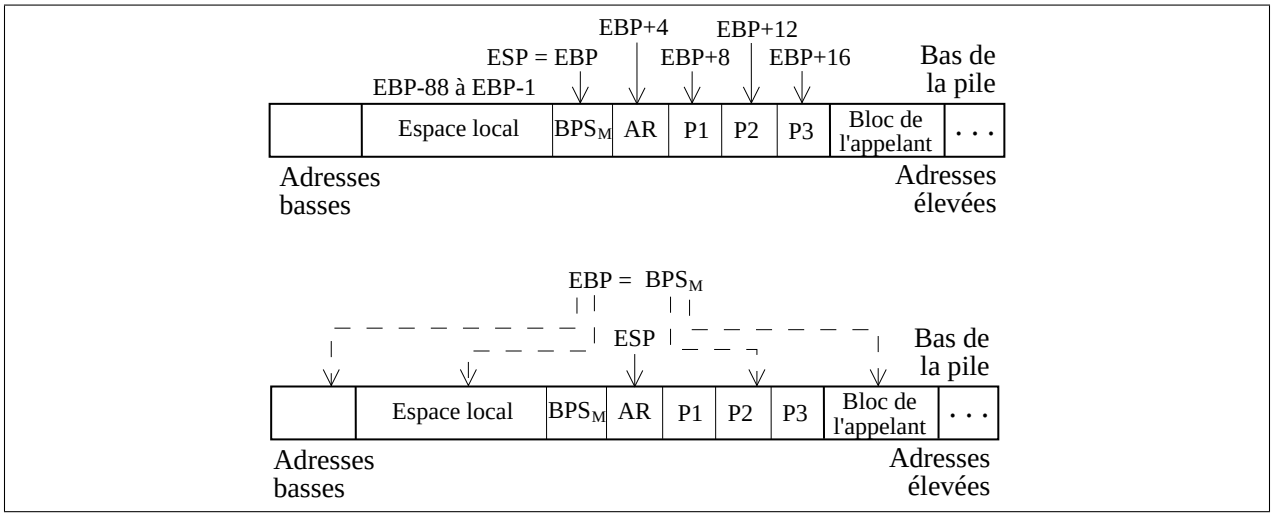

Figure 3.7: `leave` avec un pointeur de bloc de pile modifié. La partie du haut montre un état intermédiaire de l'instruction `leave`. Cet état n'est jamais vraiment atteint par le processeur, mais il aide à la compréhension. La partie du bas montre l'état à la fin de l'instruction `leave`. Les flèches avec des tirets représentent diverses adresses qui peuvent être prises par `EBP`.

pas été modifiée, on aurait `EBP` qui pointe sur le pointeur de bloc de pile dans le bloc de l'appelant. Puisque c'est son octet de poids faible qui a été modifé, BPS_M (et `EBP`) peut pointer au maximum à une distance de 255 octets de *BPS*. L'intervalle exact des adresses que peut prendre BPS_M est de $BPS - (BPS \bmod 256)$ à $BPS + 255 - (BPS \bmod 256)$.

La figure 3.8 montre l'état de la pile après que le retour dans `main()` soit complété. Le retour s'effectue correctement et `ESP` a la bonne valeur. Par contre `EBP` conserve sa mauvaise valeur, ce qui pose problème lorsque `main()`, l'appelant de `fonct()`, se termine.

En effet, la fonction `main()` n'est pas différente des autres et elle retourne à son appelant de la même façon que `fonct()`, c'est-à-dire avec les instructions `leave` et `ret`. La figure 3.9 montre l'effet de l'instruction `leave` sur l'état du programme. L'instruction suivante est `ret` et elle a pour effet de transférer l'exécution à l'adresse pointée par `ESP`. Pour que l'attaque réussisse, il suffit que l'adresse qui se trouve à cet endroit soit celle du code de l'attaquant. Comme dans le cas de l'écrasement de

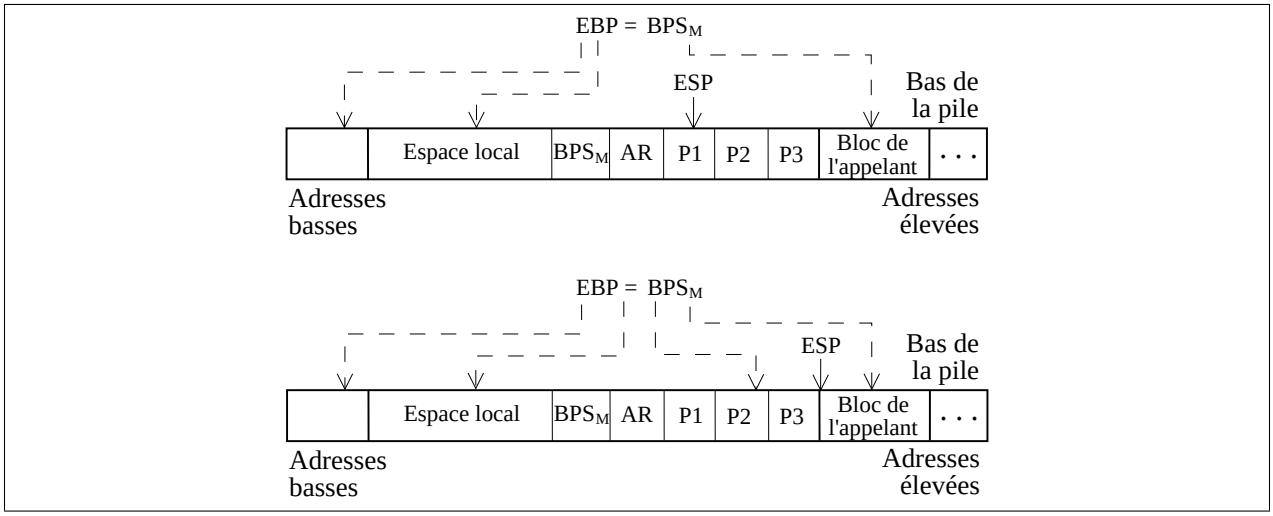

Figure 3.8: État de la pile au retour dans la procédure `main()`. Le retour est effectué correctement car `ESP` a la bonne valeur et l'adresse de retour (AR) n'a pas été modifiée. La partie du haut représente l'état du programme après l'instruction `ret`. À ce moment `EIP`, le pointeur d'instruction vaut AR. La partie du bas montre l'état du programme après l'instruction `add` qui enlève les paramètres de la pile.

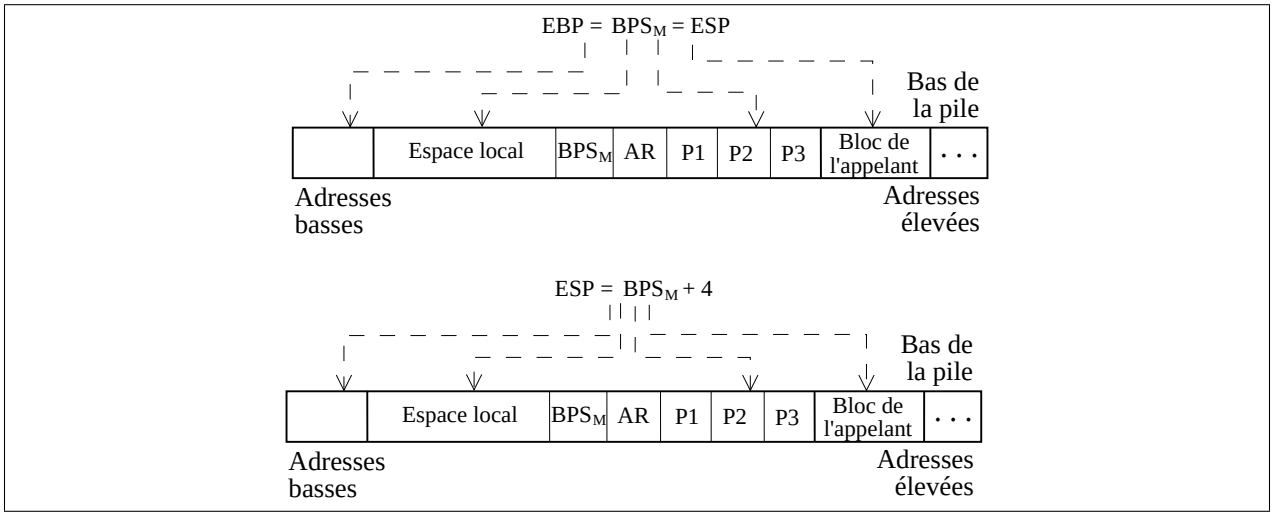

Figure 3.9: État de la pile à la toute fin de `main()`, la fonction appelante. Comme dans la figure 3.7, l'instruction `leave` a été décomposée en deux étapes pour mieux voir ce qui se passe.

l'adresse de retour, ce code peut se trouver en mémoire à peu près n'importe où, mais le plus facile pour l'attaquant est souvent de placer le code dans le tampon qui déborde.

Si on récapitule, on peut exploiter un débordement d'un seul octet en écrasant le pointeur de bloc de pile. Pour ce faire, on s'organise pour écraser l'octet de poids faible de ce pointeur de manière à pointer 4 octets avant le pointeur vers le code d'attaque. Deux instructions `leave` et deux instructions `ret` plus tard, le code s'exécuter.

Cette façon de faire est beaucoup plus sensible que l'écrasement de l'adresse de retour. En effet, en ne pouvant qu'écraser l'octet de poids faible du pointeur de bloc de pile, on n'a pas le plein contrôle sur la valeur qu'il peut prendre. Il se peut que toutes les valeurs qu'on arrive à lui faire prendre (+4) soient des adresses sur lesquelles on n'a pas de contrôle. Dans ce cas il n'est pas possible de contrôler à quelle adresse l'exécution est transférée.

Aussi, une fois de retour dans la fonction appelante, le pointeur du bloc de pile (`EBP`) n'est plus valide. Si elle ne retourne pas immédiatement, elle peut avoir un comportement étrange si elle essaie d'utiliser ses paramètres ou ses variables locales. Il est possible qu'elle fasse planter le programme. Il est aussi possible qu'en essayant de modifier une de ses variables elle modifie plutôt le code qui se trouve dans le tampon et que l'attaquant veut exécuter. Il faut tout de même savoir qu'un attaquant minutieux peut parfois construire les données qui provoquent un débordement de manière à ce que la fonction appelante voit des valeurs «crédibles» pour ses variables et ses paramètres. Il est aussi possible de placer le code à exécuter ailleurs que dans le tampon, ce qui donne plus de souplesse.

3.5.3 Écraser un pointeur de fonction

Lorsqu'un tampon est alloué statiquement ou dynamiquement sur le tas, il n'est généralement pas possible d'écraser une adresse de retour. D'autres éléments peuvent par contre être la cible d'une attaque et permettre l'exécution de code arbitraire. Parmi eux, il y a les pointeurs de fonction. Les langages C et C++ permettent en effet de déclarer un pointeur de fonction et ensuite d'appeler une fonction par ce pointeur. L'exemple 3.9 montre de quelle façon c'est fait.

Si un débordement de tampon permettait d'écraser le pointeur de fonction entre l'initialisation et l'utilisation, il serait possible de faire exécuter du code arbitraire plutôt que la fonction prévue au moment où le pointeur est utilisé. L'article [Con99] explique plus en détail comment on peut y arriver. Comme pour un débordement de tampon qui écrase une adresse de retour, le plus facile pour un attaquant est habituellement de placer son code dans le tampon qui déborde. L'adresse à donner au pointeur de fonction est alors celle du code dans ce tampon.

Contrairement à ce qui se passe généralement lorsqu'on écrase une adresse de retour, il peut y avoir beaucoup de temps qui s'écoule entre le moment où le pointeur de fonction est écrasé et le moment où il est utilisé. Ceci peut jouer contre l'attaquant car un débordement doit parfois écraser des variables importantes avant d'atteindre le pointeur de fonction recherché. Il ne faut pas que l'utilisation de ces variables fasse planter le programme, ni qu'elle modifie le code ou les données injectées pour que l'attaque réussisse.

```
#include <stdio.h>

/* Déclaration d'un pointeur de fonction */
void (*ptr_allo)(const char *nom);

void allo_francais(const char *nom)
{
        printf("Bonjour %s!\n", nom);
}

int main(int n, char **argv)
{
        /* Initialisation du pointeur de fonction */
        ptr_allo = allo_francais;
        if (argv[1])
                /* Utilisation du pointeur */
                ptr_allo(argv[1]);
        return 0;
}
```

Exemple 3.9: Appel d'une fonction par un pointeur de fonction

3.5.4 Copier et exécuter du code arbitraire par double retour

La technique présentée ici est décrite en détail dans [Woj98]. Les articles [One96, Des97a] permettent d'avoir plus d'informations sur certaines idées qu'elle utilise. Cette technique est en fait un cas particulier d'écrasement de l'adresse de retour qui a été présenté à la section 3.5.1. Ici, au lieu de retourner directement à l'adresse où se trouve le code à exécuter, on retourne à une fonction qui va copier le code qu'on veut exécuter ailleurs en mémoire puis retourner à ce code.

La première question qu'on peut se poser est pourquoi vouloir copier du code déjà en mémoire avant de l'exécuter plutôt que l'exécuter directement à l'endroit où il se trouve. La raison est que certaines parties de la mémoire peut ne pas être exécutable. En particulier, certains systèmes d'exploitation ne permettent pas l'exécution de code sur la pile. La technique présentée ici permet à un attaquant de copier de la pile au tas le code qu'il a préalablement injecté dans un programme.

Pour comprendre le fonctionnement, il faut se souvenir de l'organisation de la pile attendue à l'entrée d'une fonction. La figure 3.10 montre cette organisation. Normalement, la fonction appelante

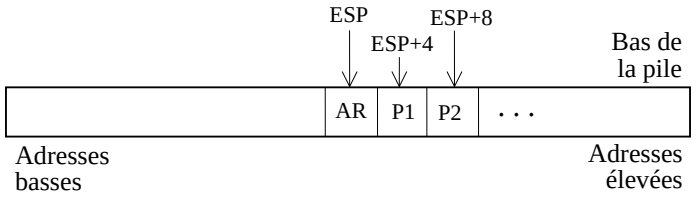

Figure 3.10: Pile à l'entrée d'une fonction

empile d'abord les paramètres, puis elle passe le contrôle à la fonction appelée en même temps qu'elle

empile l'adresse de retour à l'aide de l'instruction `call`.

Ici c'est plutôt au moyen d'une instruction `ret` qu'on veut appeler une fonction qui copiera le code, par exemple `strcpy()`. Juste avant l'exécution de cette instruction, la pile doit donc ressembler à la figure 3.11. Il faut bien comprendre que dans cette figure, *AA* est l'adresse de la fonction que

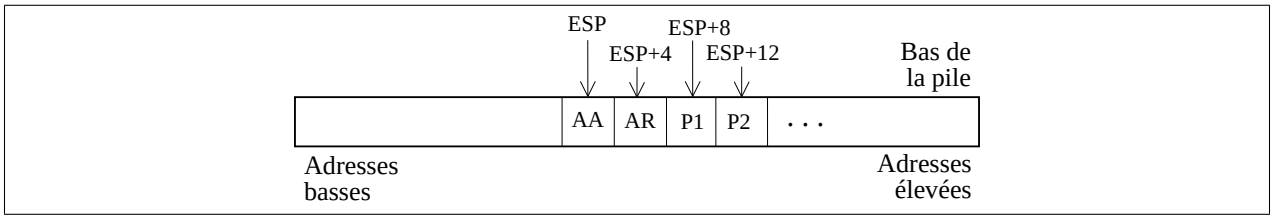

Figure 3.11: Pile avant un «appel par `ret`». *AA* représente l'adresse de la fonction à appeler et *AR* représente l'adresse à laquelle cette fonction doit retourner une fois qu'elle a terminé son travail.

l'on veut appeler, mais pour la fonction qui exécute l'instruction `ret`, c'est aussi l'adresse de retour.

En théorie, pour copier notre code et l'exécuter, il suffit d'utiliser :
- pour *AA* : l'adresse de `strcpy()` ;
- pour *AR* : l'adresse d'une zone de mémoire quelconque qui est accessible en écriture et exécutable ;
- pour *P1* : la même adresse que pour *AR* ;
- pour *P2* : l'adresse du code à exécuter déjà en mémoire.

Pour *P2* il n'y a pas de problème si l'adresse ne contient pas de 0. Dans le cas contraire, il faut chercher à placer ailleurs en mémoire le code à exécuter. Pour *P1* et *AR* on peut choisir à peu près n'importe quelle adresse valide de la zone de données qui ne contient pas de 0, donc ça devrait être facile. Il faut remarquer qu'il est très rare que l'espace des données n'est pas exécutable.

Si on cherche tant à éviter les 0, c'est que les débordements de tampon se produisent habituellement dans l'utilisation incorrecte des fonctions de manipulation de chaînes de caractères. Si un 0 se glisse dans la chaîne qui tente d'exploiter un débordement, la copie se termine à cet endroit et les données qui suivent, essentielles pour exploiter «correctement» le débordement, sont ignorées.

Ça peut être plus compliqué pour *AA* qui doit prendre pour valeur l'adresse de `strcpy()`. C'est parce que, par mesure de défense contre les attaques de ce type, la retouche[5] pour obtenir une pile non exécutable du Openwall Project [Des02] modifie l'adresse à laquelle la bibliothèque C est chargée en mémoire pour qu'elle soit sous 01000000_{16}. De cette façon, toutes les adresses qui pointent sur des fonctions de la bibliothèque C contiennent au moins un octet à 0 et il est impossible de les utiliser si le débordement se fait à l'aide d'une fonction qui traite une chaîne de caractères.

Heureusement (ou malheureusement, selon le point de vue !) on n'est pas obligé d'utiliser directement cette adresse. En effet, si la bibliothèque C peut être chargée à une adresse avec l'octet de poids fort à 0, ce n'est pas le cas pour le programme principal, qui est habituellement chargé à une adresse qui a l'octet de poids fort égal à 8. Le code et les données du programme sont chargés à cet endroit. Une partie du code forme la table des liens de procédure, ou PLT[6].

Pour comprendre à quoi sert la PLT, il faut savoir comment fonctionnent l'édition des liens en présence de bibliothèques partagées sous Linux. L'édition des liens pour les références à une

[5] Retouche est utilisé à la place du terme anglais *patch*.
[6] *Procedure Linking Table*

bibliothèque partagée ne se fait pas lors de la compilation d'un programme, mais lors de l'exécution (édition des liens dynamiques). En effet, lorsqu'un programme effectue un appel à une fonction qui se trouve dans une bibliothèque partagée, l'éditeur des liens ne sait pas à quelle adresse la bibliothèque partagée sera chargée lors de l'exécution, ni à quel endroit dans cette bibliothèque se trouvera la fonction appelée. Il crée donc une entrée dans la PLT pour cette fonction. L'exemple 3.10 donne un aperçu du fonctionnement de l'appel d'une fonction dans une bibliothèque partagée.

Voici le code correspondant à l'appel de la fonction `strcpy()` dans la bibliothèque C lorsqu'elle est utilisée sous forme de bibliothèque partagée :

```
0x80483f9 <main+9>:     push   $0x8048474
0x80483fe <main+14>:    push   $0x80495c0
0x8048403 <main+19>:    call   0x8048300 <strcpy>
```

On voit que l'appel n'est pas effectué directement dans la bibliothèque C, mais plutôt dans la PLT.

```
0x8048300 <strcpy>:     jmp    *0x8049584
0x8048306 <strcpy+6>:   push   $0x18
0x804830b <strcpy+11>:  jmp    0x80482c0 <_init+40>
```

Si on examine ce qui se trouve à l'adresse 0x08049584, on voit que la valeur correspond à l'adresse de l'instruction suivante, soit le `push`. Cette table des adresses est appelée GOT, pour *Global Offset Table*.

```
0x8049584 <_GLOBAL_OFFSET_TABLE_+24>:   0x08048306
```

L'instruction `jmp` suivante donne le contrôle à l'éditeur des liens dynamiques. Ce dernier modifie l'entrée de `strcpy()` dans la GOT pour qu'elle pointe directement sur `strcpy()` dans la bibliothèque C.

```
0x8049584 <_GLOBAL_OFFSET_TABLE_+24>:   0x001a4120
```

De cette manière, au prochain appel de `strcpy()` via la PLT, la première instruction `jmp` donnera le contrôle directement à la bibliothèque C plutôt que de passer par l'éditeur des liens dynamiques. On remarque que Linux s'exécute avec la retouche pour la pile non exécutable du Openwall Project parce que l'octet de poids fort de l'adresse vaut 0.

Exemple 3.10: Appel d'une fonction dans une bibliothèque partagée

Cet exemple démontre bien qu'il est possible d'appeler une fonction de la bibliothèque C sans y pointer directement. Passer par la PLT permet d'éviter d'avoir à travailler avec des octets nuls. Pour qu'une fonction de la bibliothèque C ait une entrée dans la PLT, il faut simplement qu'elle soit appelée à un endroit dans le programme.

La fonction `strcpy()` n'est pas la seule fonction qui peut être utilisée pour copier du code d'un endroit à un autre. `strncpy()`, `sprintf()`, `wcscpy()` et `memmove()` ne sont que quelques-unes des fonctions de la bibliothèque C qui le permettent.

Pour résumer cette section, nous avons vu une technique qui permet d'exécuter du code arbitraire même en présence de deux mécanismes de protection contre ce genre d'attaque. Ces deux mécanismes

de protection seront présenter plus en détail au chapitre 5.

3.5.5 Écraser un pointeur puis la structure de atexit()

La technique présentée dans cette section est décrite plus en détail et avec d'autres techniques similaires dans [BK00]. Ici le but d'exploiter un programme même lorsque l'adresse de retour est protégée contre l'écrasement. Le chapitre 5 décrit des méthodes qui permettent de détecter ou d'éviter l'écrasement de l'adresse de retour. La technique présentée à la section 3.5.1 ne peut alors pas être utilisée. Nous avons vu à la section 3.5.3 qu'il était parfois possible d'écraser un pointeur de fonction. La technique présentée ici est une extension de cette technique lorsqu'un débordement de tampon ne permet pas d'écraser un pointeur de fonction, mais plutôt un pointeur de données.

Plusieurs conditions doivent être réunies pour que cette technique puisse être exploitée par un attaquant. Ces conditions sont les suivantes.

- Il doit y avoir un débordement de tampon ! ;-)
- Un pointeur doit pouvoir être écrasé lors du débordement.
- Le pointeur doit être utilisé comme destination d'une opération de copie après le débordement.
- Le pointeur ne doit pas être initialisé entre le débordement et la copie.
- L'attaquant doit avoir le contrôle sur les données qui sont copiées.

Le programme présenté à l'exemple 3.11 répond à ces conditions. Ce programme ne ressemble certainement pas à un programme réel. On dirait plutôt qu'il est conçu spécifiquement pour être

```
#include <string.h>

int main(int n, char **argv)
{
        char *p;
        char tampon[20];

        p = tampon;

        strcpy(p, argv[1]);

        strncpy(p, argv[2], sizeof(tampon));

        return 0;
}
```

Exemple 3.11: Programme où on peut écraser un pointeur

exploité ! Il en est ainsi pour que le fonctionnement de l'attaque soit plus clair. Dans [BK00], des exemples plus convaincants de programmes vulnérables sont donnés. Il faut surtout remarquer qu'il peut se passer à peu près n'importe quoi entre le `strcpy()` et le `strncpy()`, à condition que p ne soit pas réinitialisé. Il faut aussi remarquer que `strcpy()` peut être remplacé par n'importe quelle fonction qui provoquent un débordement de `tampon` et que `strncpy()` peut être remplacé par n'importe quelle fonction de copie.

L'attaque se réalise en deux étapes. Dans un premier temps le débordement de tampon permet d'écraser le pointeur. On lui donne la valeur de l'adresse d'un pointeur de fonction qui sera utilisé plus tard. Ensuite, lors d'une opération de copie, c'est le pointeur de fonction qui est écrasé pour pointer vers le code de l'attaquant. Il ne reste plus qu'à attendre que le pointeur soit utilisé dans un appel de fonction.

Pour la première étape, il faut connaître l'adresse d'un pointeur qui sera utilisé pour l'appel d'une fonction. Dans [BK00], on suggère entre autres d'écraser un pointeur de fonction des structures utilisées par `atexit()`. Cette fonction permet d'exécuter d'autres fonctions au moment où le programme se termine. En écrasant un pointeur de fonction qui est sauvegardé dans ses structures, il est donc possible de faire exécuter du code supplémentaire en supposant que le programme se termine normalement. Il faut savoir que deux fonctions sont enregistrées automatiquement au début de l'exécution d'un programme pour qu'elles soient exécutées à la fin, une pour le ménage du programme principal et l'autre pour celui de la bibliothèque C.

Dans [BK00], c'est le symbole *fnlist* qui est utilisé pour retrouver la structure. Il a été remplacé par *initial* dans glibc, mais il n'est pas visible car il n'est pas exporté. On peut retrouver son adresse en suivant l'exécution de `atexit()` et en comparant avec le code source de `glibc`. L'exemple 3.12 explique plus en détail cette opération.

Il faut d'abord identifier le code qui accède à la structure. C'est *__exit_funcs* qui contient un pointeur vers *initial* au début de l'exécution du programme.

```
0x4004cf7a <__cxa_atexit+122>:  mov    0x8c0(%ebx),%eax
0x4004cf80 <__cxa_atexit+128>:  mov    (%eax),%esi
0x4004cf82 <__cxa_atexit+130>:  test   %esi,%esi
```

Ici `eax` contient l'adresse de *__exit_funcs* et `esi` contient l'adresse de *initial*.

```
(gdb) info reg eax esi
eax            0x40134d1c       1075006748
esi            0x40139e40       1075027520

(gdb) x/1 $eax
0x40134d1c <__fpu_control+28>:  0x40139e40
(gdb) x/12 $esi
0x40139e40 <errno+928>: 0x00000000  0x00000002  0x00000004  0x40009e50
0x40139e50 <errno+944>: 0x00000000  0x00000000  0x00000004  0x08048500
0x40139e60 <errno+960>: 0x00000000  0x00000000  0x00000000  0x00000000
```

On peut donc écraser le pointeur qui se trouve à 0x40139e40 + 12 ou à 0x40139e40 + 28.

Exemple 3.12: Adresse de la structure utilisée par `atexit()`

Lorsque le pointeur de fonction que l'on veut écraser est identifié, c'est avec son adresse qu'on écrase d'abord le pointeur qui sert de destination pour une opération de copie subséquente. La première flèche de la figure 3.12 indique le résultat qu'on cherche à obtenir suite à cette opération.

Pour la deuxième étape, le pointeur que l'ont vient d'écraser est réutilisé comme destination d'une opération de copie. Ici, même si le programme effectue une vérification des bornes, ça n'empêche pas

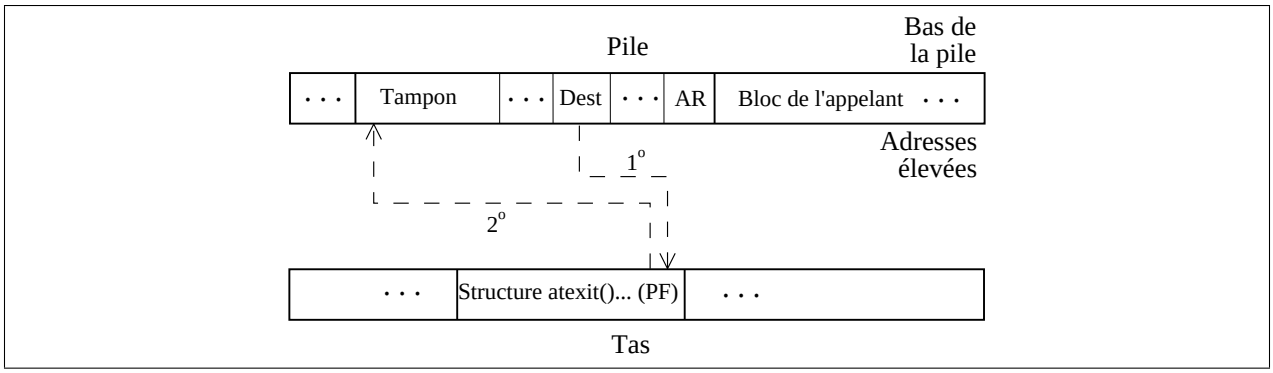

Figure 3.12: Pour écraser le pointeur de fonction *PF*, il faut d'abord écraser le pointeur *dest* d'une opération de copie. Les flèches représentent les pointeurs. Dans cette figure, le code que l'attaquant cherche à exécuter se trouve donc dans le tampon qui déborde.

d'exploiter le programme puisqu'on veut seulement écraser un pointeur qui occupe 4 octets. C'est parce que le pointeur ne pointe plus sur ce que le programmeur croit qu'il pointe. La deuxième flèche de la figure 3.12 montre le résultat de cette opération. La source de données pour la copie doit bien sûr être sous le contrôle de l'attaquant pour qu'il puisse être en mesure de donner la valeur qu'il veut au pointeur de fonction. Elle n'est pas représentée dans la figure 3.12.

Pour résumer cette section, nous avons vu une attaque qui permet de déjouer une autre technique de protection contre les débordements de tampon. Bien qu'une telle attaque ne soit pas applicable pour tous les programmes, elle démontre que la technique de protection n'est pas à toute épreuve.

3.5.6 Autres techniques pour exécuter du code arbitraire

Il existe plusieurs autres techniques qui permettent l'exécution de code arbitraire. Nous en présentons ici quelques-unes sommairement avec les références qui permettent d'en savoir plus.

3.5.6.1 Écraser le pointeur de la table des méthodes virtuelles

En C++, les adresses des méthodes virtuelles d'une classe sont conservées dans une table (*vtable*) et un pointeur vers cette table se trouve dans chaque instance de la classe. Il est parfois possible d'écraser ce pointeur pour qu'il pointe sur une table créée par l'attaquant. De cette façon il lui est possible de faire exécuter le code qu'il choisit lors d'un appel suivant à une méthode virtuelle.

Cette technique est décrite en détail dans [rix00]. Cet article donne de plus un exemple où le programme qui est attaqué continue son exécution normalement après que l'attaque soit terminée.

3.5.6.2 Écraser un jmp_buf

Dans [Con99], une variante de l'écrasement d'un pointeur de fonction est suggérée. Cette variante consiste à écraser le pointeur d'instruction qui est sauvegardé dans un `jmp_buf`. Cette structure est utilisée par les fonctions `setjmp()` et `longjmp()` de la bibliothèque C. Elles permettent de conserver

une partie de l'état d'un programme, y compris le pointeur de pile et le pointeur d'instructions, et d'y retourner plus tard. Elles sont surtout utilisées pour effectuer de la gestion d'erreurs.

3.5.6.3 Écraser un pointeur puis l'adresse de retour

Dans [BK00], une variante de la technique mentionnée à la section 3.5.5 est mentionnée. Au lieu d'écraser un pointeur de fonction de `atexit()`, il est possible d'écraser directement l'adresse de retour de la fonction. L'idée est qu'en écrasant directement l'adresse de retour plutôt que de l'écraser à partir du tampon, on évite d'être détecté par les mesures de protection du compilateur StackGuard. Il faut dire qu'il existe au moins une version de ce compilateur qui est immunisée contre ce genre d'attaque [Imm00]. Le chapitre 5 donne plus d'informations sur le fonctionnement de StackGuard.

3.5.6.4 Écraser un pointeur puis la GOT

Toujours dans [BK00], une autre variante est expliquée. Cette dernière demande à peut près les mêmes conditions que les deux autres pour être appliquée, mais elle a la particularité de permettre de déjouer à la fois les mesures de protection des compilateurs StackGuard et Stack Shield ainsi que celles de la retouche empêchant l'exécution de code sur la pile.

La technique consiste à écraser l'entrée de la GOT qui correspond à une fonction de la bibliothèque C appelée juste après l'opération qui permet d'écraser la GOT. On y copie un pointeur vers le code que l'on place juste après l'entrée de la GOT qu'on écrase. Tout ça peut donc être copié en même temps.

3.5.6.5 Écraser le pointeur `__exit_funcs`

Dans [Bou00a], on remarque qu'on peut parfois écraser le pointeur vers la structure utilisée par `atexit()` lorsque la bibliothèque C est liées statiquement. On explique de plus que cette structure correspond parfaitement avec la structure utilisée par le système d'exploitation pour le passage d'arguments à un programme. Il faut toutefois savoir que la structure utilisée par `atexit()` dans glibc est différente de celle décrite dans l'article, et que la technique n'est probablement pas applicable. Ce qui est décrit dans l'article semble venir de BSD.

3.6 Autres types d'attaques

Dans cette section, nous décrivons sommairement certains types d'attaques qui, sans permettre l'exécution de code arbitraire, sont très souvent tout aussi dangereuses que celles qui le permettent. En effet, la plupart des programmes contiennent déjà beaucoup de code qui peut être «mal» utilisé. Les techniques suivantes expliquent comment ce code peut être exploité.

3.6.1 Retourner dans la bibliothèque C

Dans [Des97a], Solar Designer détaille une méthode permettant de déjouer la protection offerte par sa propre retouche empêchant l'exécution de code sur la pile (maintenant devenue le Openwall Project). Nous avons mentionné cet article à la section 3.5.4. Le principe est de modifier l'adresse de retour pour pointer sur une fonction de la bibliothèque C au lieu de la pile. On modifie également les mots suivants de la pile pour que la fonction voie des paramètres «intéressants».

L'application la plus courante de cette technique est l'exécution de l'interpréteur de commandes /bin/sh par la fonction `system()`. Ceci est possible sans avoir à injecter `"/bin/sh"` en mémoire puisque la bibliothèque C contient déjà cette chaîne de caractère en mémoire. L'exécution d'un interpréteur de commandes (*shell*) permet à l'attaquant de faire à peu près tout ce qu'il veut par la suite.

L'article mentionne aussi qu'il est possible d'appeler deux fonctions de suite lorsque la première ne prend qu'un seul paramètre.

Le même article donne aussi une technique de protection contre ce type d'attaque, le chargement en mémoire à une adresse qui comporte un octet nul. Cette technique est expliquée dans le chapitre 5, mais nous avons vu à la section 3.5.4 qu'elle pouvait être déjouée.

3.6.2 Retourner dans la PLT, écraser la GOT, retourner à system()

Dans [Woj98], on explique une autre façon d'exploiter un programme lorsqu'il s'exécute sous la protection de la retouche empêchant l'exécution de code sur la pile du Openwall Project. À la différence de la technique expliquée à section précédente, cette méthode fonctionne dans le cas où la bibliothèque C est chargée à une adresse comportant un octet nul. À la différence de la méthode de la section 3.5.4, elle fonctionne également si les données sont non exécutables. Par contre, elle ne permet pas l'exécution de code arbitraire.

La technique consiste à modifier l'adresse de retour et le mot suivant de façon à retourner deux fois de suite à l'entrée de la PLT pour `strcpy()`. De cette façon on évite les octets nuls dans l'adresse. Lors du premier appel (ou retour !) le pointeur source (2^{e} paramètre) pointe vers un nom de fichier. À la fin de ce nom de fichier se trouve l'adresse de l'entrée de `system()` dans la bibliothèque C. Le pointeur destination (1^{er} paramètre) est l'adresse de l'entrée de la GOT pour `strcpy()`, mais ajusté pour que cette entrée soit écrasée avec l'adresse de `system()` qui est à la fin du nom de fichier.

Lors du deuxième appel (ou retour !) c'est donc `system()` qui est appelé au lieu de `strcpy()`. Son unique paramètre est le nom du fichier à exécuter que l'attaquant a pris soit de placer dans un répertoire temporaire au préalable.

On peut se demander pourquoi on voudrait retourner deux fois dans `strcpy()` d'une façon aussi complexe pour en fin de compte appeler `system()`. Pourquoi ne pas retourner directement à `system()` en passant par son entrée de la PLT ? Le problème est que si le programme que l'on exploite n'utilise pas `system()`, cette fonction n'a pas d'entrée dans la PLT. Par contre, elle est toujours présente dans la bibliothèque C.

Une autre question qu'on peut se poser est comment est-ce qu'on peut donner une adresse qui comporte un octet nul dans le nom d'un fichier, mais pas directement comme adresse de retour.

C'est parce que c'est l'octet de poids fort qui est nul, donc le dernier lorsqu'on utilise un processeur qui travaille en mode petit-boutiste. Puisqu'on veut placer un paramètre après l'adresse de retour, il n'est généralement pas possible de placer un octet nul à l'adresse de retour.

3.6.2.1 Écraser un pointeur puis la GOT pour system()

Dans [BK00], on donne une autre technique qui permet de déjouer plusieurs méthodes de protection contre les débordements de tampons. Comme à la section 3.5.6.4, plusieurs conditions doivent être réunies pour que cette technique soit applicable. Elle consiste à d'abord écraser un pointeur destination d'une opération de copie, puis écraser avec l'adresse de `system()` l'entrée de la GOT pour une fonction qui est appelée juste après. Cette fonction doit prendre en paramètre une chaîne de caractères et l'attaquant s'organise pour que la chaîne corresponde à une commande exécutable.

Chapitre 4

Conséquences des vulnérabilités de chaîne de format

Nous avons vu à la section 2.6 que les spécificateurs de conversion des chaînes de format demandent à une fonction d'interpréter d'une certaine façon un paramètre supplémentaire. L'absence de vérification de correspondance entre les paramètres utilisés par une fonction et ceux passés par son appelant de même que certaines fonctionnalités surprenantes offertes par les chaînes de format peuvent donner lieu à des vulnérabilités au moins aussi grandes que celles provoquées par un débordement de tampon.

Ce chapitre est basé en grande partie sur l'information se trouvant dans [scu01, por01, Thu01, New00].

4.1 Obtenir des données

Nous savons déjà qu'il est possible pour l'utilisateur d'obtenir le contenu de la pile lorsqu'il peut voir la chaîne formatée correspondant à une chaîne de format arbitraire. Pour ce faire, il peut par exemple utiliser des spécificateurs de conversion «`%X`». Nous avons de plus vu que, grâce au spécificateur de conversion «`%s`», il est possible d'obtenir le contenu de la mémoire à partir de n'importe quelle adresse se trouvant sur la pile et jusqu'au caractère nul suivant.

Sur la pile se trouvent les paramètres des fonctions en cours d'appel en plus de leurs variables locales. Permettre à l'utilisateur d'accéder à cette information peut donc être un problème de confidentialité important.

4.1.1 Accéder efficacement à la pile

Donner un spécificateur de conversion dans la chaîne de format pour chaque mot de la pile n'est pas très optimal. En effet, les programmes limitent habituellement la taille des tampons qui servent à stocker la chaîne de format et la chaîne formatée. Le nombre de spécificateur de conversion qu'on peut placer dans une chaîne de format est donc limité. En supposant qu'il n'y a pas de limite sur

la chaîne de format, la chaîne formaté a habituellement une taille maximale. Une fois que la chaîne formatée a atteint cette limite, les spécificateurs de conversion supplémentaires ne permettent pas d'obtenir de l'information qui se trouve plus loin sur la pile. On cherche donc à minimiser la taille de la chaîne de format et celle de la chaîne formatée.

Si c'est la chaîne de format qui est limitée, il est possible d'utiliser«`%f`», `%F`», «`%e`», «`%E`», «`%g`» «`%G`», `%a`» ou «`%A`» pour passer 8 octets de la pile au lieu de 4 octets par spécificateur de conversion. L'inconvénient est que la taille de l'information en sortie peut être très grande. On peut la limiter à 7 caractères au maximum par spécificateur de conversion en utilisant par exemple «`%.e`», mais au prix d'une chaîne de format plus longue. Le spécificateur de conversion «`%.f`» peut parfois être utile pour qu'une conversion génère moins de caractères en sortie. Lorsqu'on l'utilise et que la valeur à convertir est dans l'intervalle $[0, 9.5[$, ce qui est en général assez fréquent, un seul caractère est généré en sorti. Par contre, dans le pire des cas, jusqu'à 310 caractères peuvent être générés.

Si c'est la chaîne formatée qui a une taille trop restreinte, on peut utiliser «`%c`» pour n'avoir qu'un caractère en sortie par spécificateur de conversion. L'inconvénient majeur de cette technique est le risque assez élevé de placer des caractères nuls dans la chaîne formatée. Tout ce qui suit un caractère nul risque de ne pas être retourné à l'utilisateur. Lorsqu'on sait qu'une valeur sur la pile est 0, il est possible d'utiliser «`%d`» qui donne lui aussi un seul caractère en sortie pour ce cas, mais ça ne règle pas le problème pour toutes les valeurs qui sont multiples de 256 et qui correspondent aussi au caractère nul.

Plutôt que de spécifier une largeur de champ dans un spécificateur de format, comme dans «`%10u`», il est possible d'utiliser le symbole «`*`» qui signifie d'utiliser le «paramètre» suivant comme longueur de champ. En général, ça donne une chaîne formatée beaucoup plus grande, mais ça permet de passer 4 octets supplémentaires de la pile avec un seul caractère supplémentaire dans la chaîne de format. Certaines versions de la bibliothèque C permettent de placer plusieurs `*` à la place de la longueur du champ. Pour chacun d'eux, 4 octets de la pile sont passé et seul le dernier compte pour la longueur du champ. Il est de plus possible de spécifier une autre largeur de champ à la suite des `*` et elle a priorité. On peut donc utiliser un spécificateur de format du genre «`%*****10u`» pour passer 5 paramètres et afficher le 6^{e} avec une largeur de 10. Cette technique fonctionne sur les dérivés de BSD, mais pas avec la bibliothèque utilisée sur la plupart des distributions Linux.

Avec la plupart des bibliothèques C, il est possible d'accéder directement à un paramètre par son numéro. On peut par exemple spécifier «`%12$u`» pour accéder au 12^{e} paramètre. Cette façon de faire permet d'économiser de l'espace dans la chaîne de format et dans la chaîne formatée. Certaines bibliothèques limitent le nombre de paramètres qui peuvent être accédés de cette façon, par exemple à 30 et parfois moins. Avec glibc sous Linux, il est possible d'accéder au moins aux 250 000 premiers paramètres.

4.1.2 Accéder au reste de la mémoire

Nous allons voir dans cette section que les vulnérabilités de chaîne de format permettent de voir bien plus que le contenu de la pile. Elles permettent la plupart du temps d'accéder à la totalité de l'espace d'adressage du programme.

Pour ce faire, l'utilisateur doit placer sur la pile un pointeur vers l'adresse mémoire qu'il veut obtenir. Il utilise ensuite le spécificateur de conversion «`%s`», ou «`%ls`» si le système est configuré

pour l'accepter, pour obtenir ce qui se trouve à cet endroit en mémoire et tout ce qui suit jusqu'au caractère nul suivant.

Pour placer un pointeur sur la pile, l'utilisateur peut donner à un entier une valeur qui correspond à une adresse. Il peut aussi placer les pointeurs dans la chaîne de format si le tampon qui la contient se trouve sur la pile. De cette façon, il peut spécifier du même coup plusieurs adresses et ainsi obtenir une plus grande partie de la mémoire en une opération.

Certains caractères nécessaires pour former les adresses ne peuvent pas être saisies au clavier. On peut cependant les générer pour les mettre dans un fichier au moyen de la commande `echo` ou dans un programme C avec la fonction `printf()` en utilisant une séquence d'échappement hexadécimale. Par exemple, le pointeur à l'adresse 0x12345678 peut être générer avec la chaîne `"\x78\x56\x34\x12"` pour un processeur petit-boutiste. Il faut noter que de cette façon il n'est généralement pas possible de générer un octet avec la valeur 0 puisque le caractère nul sert à terminer une chaîne de caractères de façon générale. Le caractère «`%`» peut aussi causer des problèmes parce qu'il a une signification particulière dans les chaînes de format.

Une fois que le pointeur est placé sur la pile, il faut être en mesure de l'atteindre puis d'utiliser le spécificateur de conversion qui permet d'obtenir la zone de mémoire désirée. Pour ce faire, on peut utiliser une des techniques présentées à la section 4.1.1. Il est souvent possible de déclencher une vulnérabilité de chaîne de format plusieurs fois dans l'exécution d'un programme. Sinon, il est parfois possible d'exécuter le programme à plusieurs reprises. De cette façon on peut obtenir le contenu de la mémoire du programme en plusieurs étapes.

4.2 Écraser des données

Nous avons déjà vu certaines possibilités offertes par les chaînes de format qui peuvent paraître surprenantes, comme «`*`» et «`$`». Nous allons voir ici que le spécificateur de conversion «`%n`» est encore plus intéressant que tout ce qui précède. Il permet de compter le nombre de caractères «convertis» jusqu'à un certain point dans la chaîne formatée. Le résultat est écrit en mémoire à l'adresse donnée par le paramètre suivant. L'exemple 4.1 montre l'utilisation du spécificateur de conversion «`%n`».

Il faut noter qu'il n'y a pas de débordement de tampon dans cet exemple car on utilise la fonction `snprintf()` en prenant soin de bien spécifier la taille du tampon. On voit par contre que les caractères «convertis» comptent même s'ils ne sont pas inscrits dans le tampon de la chaîne formatée à cause d'un manque d'espace. Le spécificateur «`%n`» permet donc de calculer la taille exacte qu'occupe une chaîne une fois formatée en spécifiant un très petit tampon. On peut ensuite réserver un tampon de la bonne taille et appeler la fonction avec la même chaîne de format, les mêmes paramètres et ce nouveau tampon comme destination pour obtenir la chaîne formatée complète.

Avec les techniques qui sont présentées à la section 4.1.2, on comprend tout de suite qu'il est possible d'écraser un ou plusieurs mot en mémoire. Ce qui est peut-être moins évident est qu'on peut y placer une valeur arbitraire. Ici la taille du tampon de la chaîne formatée n'est plus un facteur limitatif comme c'était le cas lorsqu'on voulait obtenir une partie de la mémoire. La chaîne de format est par contre habituellement limitée.

```
#include <stdio.h>

int main()
{
        char tampon[10];
        int n=0;
        snprintf(tampon, sizeof(tampon),
                 "Une chaîne %s 30 caractères...%n", "de", &n);
        printf("%d caractères auraient dû être inscrits.\n", n);
        return 0;
}
```

L'exécution donne :

```
$ ./format2
30 caractères auraient dus être inscrits.
```

Exemple 4.1: Utilisation de «`%n`» dans une chaîne de format

Un «`%n`» correspond à un pointeur vers un `int`. Un `int` a le plus souvent une taille de 32 bits. Il serait irréaliste de vouloir passer une chaîne de format contenant jusqu'à 4 milliards de caractères pour écraser un mot de 32 bits avec une valeur arbitraire. On pourrait penser utiliser une grande valeur pour la largeur du champ, de façon à augmenter le nombre de caractères formatés sans avoir à les spécifier dans la chaîne de format. Par exemple, la chaîne de format `"%1234567890u%n"` devrait placer la valeur 1 234 567 890 dans l'entier pointé par le deuxième paramètre. Ça ne fonctionne pas parce que la bibliothèque C n'est pas faite pour gérer des champs aussi grands.

4.2.1 Diviser pour régner

Ce qu'on peut faire par contre est d'utiliser «`%hn`». Le drapeau «`h`» indique que le paramètre est un pointeur vers un `short` plutôt qu'un `int`. De cette façon seulement la partie de poids faible du compteur de caractères est inscrite en mémoire dans un mot de 16 bits. La largeur maximale à spécifier entre les «`%n`» est ainsi environ 2^{16}, mais dans tous les cas inférieurs à 66 000, ce qui est raisonnable pour la plupart des bibliothèques C. Il faut donc deux «`%hn`» et donc deux pointeurs sur des mots de 16 bits qui se suivent en mémoire pour écraser un mot de 32 bits.

Pour être en mesure d'écrire une valeur arbitraire en mémoire, il faut savoir à chaque «`%hn`» le nombre exact de caractères formatés. Par exemple, lorsqu'on spécifie la largeur d'un champ pour un entier non signer avec «`%u`», il faut que la largeur soit au moins de 10. Si on spécifiait seulement une largeur de 9 caractères et que la valeur du paramètre était de 1 000 000 000 ou plus, il y aurait 10 chiffres en sortie et les calculs ne seraient plus bons.

Pour inscrire un mot de 16 bits de valeur v quand on sait qu'il y a déjà n caractères formatés au point où on est rendu dans la chaîne de format, il faut lui ajouter `"%`m`u%hn"` où m est calculé de la façon suivante :

$$m = (2^{16+1} + v - 10 - n \bmod 2^{16}) \bmod 2^{16} + 10$$

Après cette partie de la chaîne de format, il y a $n + m$ caractères formatés.

Par exemple, si une chaîne de format donne 123456 caractères formatés et qu'on veut placer la valeur 34567 dans le mot de 16 bits pointé par le 2e paramètre suivant sur la pile (le premier est utilisé pour le spécificateur de format «`%u`») on trouve :

$$m = (131072 + 34567 - 10 - 123456 \bmod 65536) \bmod 65536 + 10$$

On a donc $m = 42183$ et il faut ajouter `"%42183u%hn"` à la chaîne de format. Après cet ajout, la chaîne de format donne 165639 caractères formatés.

Il faut remarquer qu'en procédant de cette façon on n'a pas de difficulté à placer une valeur totalement arbitraire en mémoire, les octets nuls ne posent aucun problème.

Il est aussi parfois possible d'utiliser le spécificateur de conversion «`%hhn`». De cette façon il est possible de n'écraser qu'un seul octet à la fois. Il n'y a toutefois pas beaucoup d'avantage à utiliser cette façon de faire car le «drapeau» «`hh`» n'est pas reconnu par autant de bibliothèques C que «`h`» et qu'il demande 4 écritures au lieu de 2 pour écraser un mot de 32 bits. La formule à utiliser pour calculer la largeur des champs entre les «`%hhn`» est la même que pour des mots de 16 bits, mais avec les 16 remplacés par des 8. Cette méthode fonctionne au moins avec glibc sous Linux.

4.2.2 Écrire des mots non alignés

Lorsqu'il n'est pas possible d'utiliser les spécificateur de conversion «`%hn`» ou «`%hhn`», soit parce qu'ils ne sont pas disponibles où parce qu'il est impossible de spécifier un champ assez large pour obtenir suffisamment de caractères entre eux, il est possible d'utiliser des spécificateurs de conversion «`%n`» ordinaires, mais à des adresses non alignées qui se recoupent partiellement. Ceci permet d'obtenir un résultat similaire.

Par exemple, on peut écraser en 4 étapes un mot qui se trouve à l'adresse a. À chaque étape on écrit un mot de 32 bits dont seul l'octet de poids faible est utile. À la première étape on écrit à l'adresse a. Ensuite on écrit à l'adresse $a - 1$, et ainsi de suite. Les calculs se font donc exactement comme ils le sont si on utilise un spécificateur de conversion «`%hhn`».

Si on veut écraser le mot de 32 bits à l'adresse 0x12345678 avec la valeur 0x9ABCDEF0, on peut le faire de la façon suivante :

Adresse	Valeur
0x12345678	0x000000F0
0x12345679	0x000001DE
0x1234567A	0x000002BC
0x1234567B	0x0000039A

On obtient en mémoire, à partir de l'adresse 0x12345678, la suite d'octets 0xF0, 0xDE, 0xBC, 0x9A, 0x03, 0x00, 0x00. On reconnaît la valeur 0x9ABCDEF0 en format petit-boutiste. Si le processeur fonctionnait en mode gros-boutiste, il faudrait commencer par écrire la valeur 0x9A puis passer à l'adresse $a - 1$. Dans les deux cas on écrase 3 octets supplémentaires, soit avant le mot de 32 bits visé, soit après ce dernier. Un autre inconvénient majeur de cette technique est qu'elle oblige des accès non alignés à la mémoire. Certains processeurs ne permettent pas ces opérations.

4.2.3 Atteindre la source de pointeurs

On a vu à la section 4.1.1 plusieurs techniques qui permettent d'accéder efficacement à une certaine partie de la pile. Si la source des adresses nécessaires pour permettre d'écraser un mot en mémoire se trouve dans un tampon sur la pile, il faut utiliser une de ces techniques. Cependant, les critères sont différents. À la section 4.1.1 on voulait minimiser l'espace occupée dans les tampons. Ici on cherche plutôt une méthode permettant de connaître le nombre exact de caractères formatés car, sans le nombre exact, il est impossible de donner une valeur arbitraire à un mot en mémoire.

On peut par exemple utiliser «`%10u`» qui donne toujours exactement 10 caractères formatés, mais d'autres choix sont possibles. Il faut aussi savoir que, selon la taille du tampon qui accueille la chaîne de format on peut avoir à minimiser la chaîne de format. L'accès direct aux paramètres peut devenir très utile si elle est disponible.

4.2.4 Accéder au reste de la mémoire, prise 2

On a vu à la section 4.1.2 qu'il était possible de créer un pointeur arbitraire à condition qu'il ne contienne pas d'octet nul. Avec les techniques présentées aux sections 4.2.1 et 4.2.2, il est maintenant possible de générer des octets nuls. Il est donc possible de modifier la source de pointeurs pour y insérer des octets nuls avant que les pointeurs ne soient utilisés. On peut donc de cette façon lire et écrire à des adresses totalement arbitraires !

4.3 Applications

Avec toutes ces techniques en main, les possibilités sont pratiquement illimitées. On peut par exemple faire planter un programme. Il suffit de lui demander de lire à une adresse nulle ou n'importe quelle adresse qui ne fait pas partie de l'image mémoire.

Toutes les cibles vues au chapitre 3 qu'on cherchait à écraser pour exploiter les débordements de tampons et exécuter du code arbitraire peuvent aussi être utilisées en présence d'une vulnérabilité de chaîne de format puisqu'on peut générer des adresses arbitraires. On a donc l'embarras du choix : adresse de retour, structures de `atexit()`, GOT, table des adresses virtuelles, etc. Ce ne sont pas les cibles qui manquent.

Si le tampon de la chaîne de format est assez grand ou qu'il est possible de déclencher la vulnérabilité plusieurs fois dans l'exécution du programme, il est possible d'utiliser la chaîne de format pour placer du code arbitraire ailleurs que sur la pile. Ceci permet de déjouer les cas où la pile n'est pas exécutable. Cette façon de faire permet même de placer des octets nuls dans le code, ce qui peut faciliter quelque peu l'écriture du code.

Lorsque l'utilisateur a accès à la chaîne formatée, il peut s'en servir pour obtenir les adresses exactes qui se trouvent sur la pile ou ailleurs en mémoire. Nous appelons cette technique force brute basée sur la réponse et elle est décrite dans [scu01]. L'annexe A décrit en détail une attaque qui utilise cette technique.

Lorsqu'il n'y a pas accès, il est toujours possible d'obtenir les adresses exactes au moyen d'une autre technique appelée force brute aveugle. Elle fonctionne en calculant le temps pris pour passer au travers de certaines chaînes de format. Elle n'est pas décrite en détaille, mais on en trouve un exemple sous forme de code source dans [scu01].

Lorsque l'utilisateur n'a pas accès au fichier exécutable du programme pour l'étudier, par exemple parce que le programme s'exécute sur un autre ordinateur par le réseau, il lui est possible de le reconstruire. En effet, on a vu qu'une vulnérabilité de chaîne de format permet souvent de lire la totalité de l'espace d'adressage d'un programme. Avec cette image mémoire, il est possible de reconstruire le fichier exécutable correspondant ou une très bonne approximation [Ces00].

On comprend donc que les vulnérabilités de chaîne de format peuvent être au moins aussi importantes que les débordements de tampons. Si ces derniers ne permettent que d'écraser la mémoire qui suit un tampon, les vulnérabilités de chaîne de format permettent le plus souvent de lire et d'écrire à des adresses arbitraires.

Chapitre 5

Éviter les vulnérabilités

Les solutions qui permettent d'éviter les débordements de tampons et les vulnérabilités de chaîne de format sont très variées. Elles se distinguent principalement par :

- le fait qu'elles soient ad hoc ou formelles ;
- le fait qu'elles soient dynamiques ou statiques ;
- le fait qu'elles évitent la vulnérabilité ou une conséquence de la vulnérabilité ;
- la présence de faux positifs et/ou de faux négatifs.

Ce chapitre fait un survol des différentes techniques et approches disponibles peu importe la catégorie dans laquelle elles se classent et peu importe leur efficacité.

5.1 Coder correctement

Lorsqu'un programme est correct, sans bogues, il ne contient pas de vulnérabilité chaîne de format ni de débordement de tampon. L'expérience montre qu'il serait par contre irréaliste de s'attendre à ce que des programmeurs écrivent du code sans bogues. Avec un langage comme le C, il est facile de faire des choses incorrectes. [Wag00, Fry00, CWP+99]. Des vulnérabilités dans différents programmes sont découvertes pratiquement à tous les jours[1]. Nous cherchons donc des moyens plus automatisés pour trouver les vulnérabilités ou les éviter. *Errare humanum est.*

5.2 Utiliser un langage immunisé

Certains langages, comme Java, sont immunisés contre les débordements de tampons. Malgré tout, il peut y avoir des conséquences non désirées à une tentative d'accès à des éléments en dehors des bornes des tampons, comme nous l'avons vu à la section 3.2. Certaines des approches que nous allons voir dans ce chapitre peuvent donc être utiles même pour des langages immunisés contre les débordements de tampons. Par exemple, le programme Wasp utilise l'analyse statique pour détecter statiquement des accès hors bornes aux tableaux dans les programmes Java.

[1] `http://online.securityfocus.com/archive/1`

De plus, il n'est pas toujours possible d'utiliser un autre langage que le C. Par exemple, pour certains modules système, les choix de langages sont souvent très limités. Il existe aussi des centaines de millions de lignes de code C et il ne serait pas réaliste de vouloir toutes les récrire [Whe02c, CWP+99, GBPdlHQ+02, Whe01].

Il existe aussi des dialectes du C qui ont été conçu spécialement pour empêcher certains problèmes comme les débordements de tampon. Un d'eux s'appelle Cyclone [cyc02]. Malgré leur apparentes similitude avec le langage C, ces dialectes sont en fait des langages différents et un programme existant ne peut pas être simplement recompilé pour bénéficier des avantages du langage. Nous n'avons pas étudié de façon détaillée ces langages.

5.3 Tester

Il est certainement possible de découvrir des problèmes causés par des débordements de tampons et des chaînes de format en effectuant des tests sur un programme. Il est par contre habituellement impossible d'obtenir la certitude qu'un programme ne contient pas ces problèmes uniquement par des tests. Les tests que subissent les programmes se concentrent habituellement sur les cas pour lesquels ils ont été conçus, alors que les vulnérabilités ont habituellement leur source dans les cas qui ne sont pas attendus par un programme.

Il existe tout de même des outils de tests automatiques qui peuvent être utiles pour découvrir certaines vulnérabilités dans les programmes. Par exemple, en 1995, fuzz a permis de d'identifier des problèmes d'accès à des tableaux ou des manipulations de pointeur dans pas moins de 24 programmes provenant de plusieurs systèmes UNIX différents [MKL+95]. Fuzz fonctionne en générant des données aléatoires en entrée pour le programme à tester.

5.4 Réduire les privilèges

Il est possible de limiter les conséquences d'un débordement de tampon ou d'une vulnérabilité de chaîne de format en exécutant un programme qui présente des risques avec des privilèges réduits. Pour ce faire, on utilise habituellement un compte non privilégié du système sur lequel le programme s'exécute. Cette technique est utilisée très souvent pour les programmes qui offrent des services sur Internet, notamment les serveurs Web. Par exemple, on exécute habituellement le serveur web apache avec un utilisateur non privilégié tel que «nobody».

Une autre technique consiste à scruter les actions d'un programme et intervenir, par exemple en arrêtant un programme, lorsqu'il s'écarte du droit chemin. Il y a entre autres Janus [jan00], Subterfugue [sub02] et syscalltrack [sys02] qui permettent ceci.

Cette approche n'est pas une solution au problème des débordements de tampons ni à celui des vulnérabilités de chaîne de format car elle ne les évite pas du tout. Elle ne fait que limiter les dégâts causés par le problème. De plus, elle ne peut pas être appliquée aux programmes qui, par leur nature, doivent avoir certains privilèges pour effectuer une action. Il est aussi très fréquent qu'après avoir gagné un accès non privilégié à un système distant, des nouvelles possibilités s'offrent à l'attaquant et lui permettent d'augmenter ses privilèges.

5.5 Modifier le compilateur

Les attaques qui utilisent des débordements de tampons exploitent la connaissance du fonctionnement interne des programmes générés par les compilateurs. Sans modifier le langage, il est possible de modifier la structure du code généré par le compilateur de manière à éviter certains débordements de tampons ou certaines façons de les exploiter. Cette section décrit quelques possibilités.

5.5.1 Protéger les adresses de retour

Lorsqu'il y a un débordement de tampon sur la pile, la cible la plus facile pour un attaquant est certainement l'adresse de retour. Elle est toujours présente et elle est (presque) toujours utilisée à la fin de la fonction. On comprend donc que certains compilateurs s'attardent à protéger spécifiquement l'adresse de retour des fonctions. Les différentes méthodes de protection discutées dans cette section sont expliquées dans [CPM+98, Imm00].

5.5.1.1 Utiliser un canari

Une façon de protéger l'adresse de retour est de la précéder en mémoire d'une certaine valeur appelée canari[2]. On suppose ici que la pile grandit vers le bas, comme c'est le cas le plus souvent. À l'entrée d'une fonction, un canari est placé sur la pile. À la sortie de la fonction, le canari est vérifié. S'il a été modifié, on suppose que l'adresse de retour a elle aussi été modifiée et qu'elle ne peut pas être utilisée. Le programme est arrêté.

On suppose donc que pour écraser l'adresse de retour par un débordement de tampons, l'attaquant devra d'abord écraser le canari. Pour que la protection soit efficace, l'attaquant ne doit pas être en mesure d'écraser le canari avec la même valeur. Pour éviter ceci, on peut choisir le canari de plusieurs façon.

On peut utiliser un mot de 32 bits de valeur 0, on parle alors de canari nul. Les opérations sur les chaînes de caractères ne permettent habituellement pas de copier des caractères nuls. Le débordement devrait donc être arrêté avant d'atteindre l'adresse de retour. Une variante consiste à utiliser une valeur formée de plusieurs caractères de terminaison. Par exemple, dans un mot de 32 bits, on peut placer les caractères «`\0`», «`\r`», «`\n`» et «`\xff`». Nous appelons une telle valeur un canari de terminaison. Elle permet d'augmenter les chances d'avoir un caractère qui arrête l'opération de copie.

Une autre possibilité est d'utiliser une valeur aléatoire, déterminée au début de l'exécution du programme. De cette façon l'attaquant ne peut pas connaître à l'avance la valeur qu'il doit utiliser pour écraser le canari.

L'utilisation d'un canari a été introduite par StackGuard et elle a été reprise dans d'autres compilateurs par la suite tel que Stack-Smashing Protector (SSP) [EY00] et Visual C++ 7.0 avec l'option «`/Gs`» [RWM02].

[2] Les mineurs gallois apportaient avec eux des canaris dans des cages pour détecter des conditions dangereuses. Lorsqu'un canari mourrait, il valait mieux pour eux de quitter la mine.

5.5.1.2 Protection avec l'assistance du processeur

L'utilisation d'un canari n'empêche pas l'écrasement de l'adresse de retour, elle permet seulement de le détecter avant qu'il pose problème. Avec la collaboration du système d'exploitation, il est possible de détecter tous les accès en écriture à une certaine adresse. De cette façon on peut arrêter le programme dès que le l'adresse de retour est modifiée.

Sur les processeurs Pentium et suivant de la famille IA-32, deux fonctionnalités du processeur peuvent être utilisées pour détecter l'écriture à une certaine adresse. D'abord, le processeur a 4 registres de débogage qui permettent de spécifier des adresses qui provoquent une exception lorsqu'elles sont lues ou écrites (c'est configurable).

Lorsque ces registres ne suffisent pas, il est possible de définir une page, qui a généralement 4 ko, comme étant accessible en lecture seulement. Tout accès en écriture provoque alors une exception et le système d'exploitation peut déterminer si c'est une adresse de retour qui est en voie d'être écrasée.

Le compilateur doit donc insérer le code au début et à la fin de chaque fonction pour indiquer au système d'exploitation de commencer et d'arrêter la protection d'une adresse de retour.

Les deux techniques sont très coûteuses en temps d'exécution parce qu'elles font appel au système d'exploitation, mais la protection au niveau des pages a un prix beaucoup plus élevé. Ceci est dû au fait qu'une page de 4 ko sur la pile ne contient pas seulement une adresse de retour, mais aussi des paramètres, des variables locales et possiblement des tableaux. Chaque accès en écriture à un de ces éléments provoque une exception que le système d'exploitation doit vérifier avant de redonner le contrôle au programme. L'utilisation des registres de débogage ne cause pas ce problème, mais elle ne permet pas de protéger plus de 4 adresses à la fois.

C'est aussi StackGuard qui a introduit ces techniques de protection. Pour ce faire, il utilise certaines fonctionnalités ajoutées à Linux par MemGuard. Les auteurs de StackGuard ont mesuré que la protection d'un programme réel en utilisant les registres de débogage augmente le temps d'exécution par un facteur qui peut aller jusqu'à 11. Pour la protection au niveau des pages, le facteur peut aller jusqu'à 460.

5.5.2 Placer les adresses de retour sur une pile différente

Pour empêcher les débordements de tampons sur la pile d'écraser une adresse de retour, il est possible de placer ces dernières sur une pile distincte de la pile principale du programme. Pour éviter de briser les conventions d'appel entre les fonctions, il est possible de copier l'adresse de retour sur une pile distincte à l'entrée d'une fonction et de la replacer sur la pile principale juste avant de retourner à l'appelant. C'est le principe qui est utilisé par Stack Shield [Ven00b, Ven00a].

5.5.3 Modifier l'ordre des variables sur la pile

Pour obtenir une plus grande protection, il est possible de protéger d'autres variables importantes en plus des adresses de retour. Par exemple, les pointeurs sont souvent une cible de choix dans les

attaques basées sur les débordements de tampons. Le langage C permet au compilateur de choisir l'ordre des variables locales d'une fonction. Il est donc possible pour le compilateur de modifier l'ordre de façon à ce que les pointeurs se retrouvent avant les tableaux. Un débordement de tampon ne peut alors pas écraser de pointeur.

Pour ce qui est des pointeurs passés en paramètre, ils ne peuvent pas être déplacés sans briser les conventions d'appel entre les fonctions, mais ils peuvent être copiés parmi les variables locales à l'entrée d'une fonction. La copie peut donc être protégée contre les débordements de tampons et l'original peut par la suite être ignoré.

C'est le projet Stack-Smashing Protector (SSP) qui a introduit l'idée de modifier l'ordre des variables pour augmenter la protection [EY00]. Cette approche ne peut cependant pas protéger les structures qui comportent à la fois des pointeurs et des tableaux puisque le langage C ne permet pas de modifier l'ordre des éléments d'une structure. L'approche ne permet pas non plus d'isoler les pointeurs passés parmi les paramètres variables d'une fonction.

5.5.4 Vérification des bornes à l'exécution

Cette section est surtout basée sur [McG98b, McG98a, McG99, JK97, Jon95, ABS93].

Les langages C et C++ permettent l'utilisation de pointeurs dans un contexte complètement déconnecté des déclarations auxquelles ils référencent. Ceci fait en sorte qu'il est loin d'être facile pour un compilateur d'effectuer la vérification des bornes à l'exécution. Par exemple, une fonction peut prendre en paramètre un pointeur vers un entier. Lors de l'exécution, cet entier pourra provenir d'une simple variable ou d'un tableau. Dans le premier cas, l'arithmétique sur ce pointeur donne un résultat non défini, mais dans le deuxième, le résultat est défini tant que le pointeur reste dans les bornes du tableau.

Quoi qu'il en soit, la définition de ces langages ne ferme pas la porte totalement à la vérification des bornes des tableaux. Elle spécifie des comportements qui donnent un résultat bien défini et d'autres qui donnent un résultat non défini. Un compilateur peut donc s'inspirer de ces définitions pour être plus ou moins restrictif sur les accès aux tableaux qui sont permis lors de l'exécution.

5.5.4.1 Représenter un pointeur par un tuple

Les pointeurs en C et C++ sont généralement représentés par l'adresse vers laquelle ils pointent. Cette représentation est utilisée parce qu'elle permet d'atteindre les performances les plus élevées étant donné qu'elle correspond exactement à la représentation utilisée par le processeur. Cette représentation n'est cependant pas imposée par le langage. Les compilateurs sont libres d'utiliser une représentation arbitraire, tant qu'elle permet de respecter la définition du langage.

En particulier, un pointeur peut être représenté par un tuple $<$*adresse*, *base*, *limite*$>$. Ici *adresse* est l'adresse de la variable pointée, *base* est l'adresse du premier élément du tableau et *limite* est l'adresse de fin du tableau, après le dernier élément. Nous appelons ceci un pointeur borné [3]. Si le

[3] Le terme anglais est *bounded pointer*

pointeur est vers une variable simple de type T, qui ne fait pas parti d'un tableau, on a :

$$adresse = base = limite - \texttt{sizeof}(T)$$

Au moment où le code prend un pointeur sur un élément d'un tableau, le compilateur est en mesure d'identifier le début et la fin du tableau. Il peut donc donner la bonne valeur aux trois champs qui forment le pointeur. Ensuite, peu importe si le pointeur est passé en paramètre ou retourné par une fonction, la base et la limite du tableau suivent. Lorsqu'il y a de l'arithmétique effectuée sur un pointeur, le compilateur peut donc vérifier très efficacement si le pointeur respecte les bornes de l'objet auquel il appartient.

Cette façon de faire a cependant quelques inconvénients. D'abord, plusieurs programmes assument qu'ils peuvent stocker un pointeur dans une variable de type `int`. Si les pointeurs sont représentés par un tuple de 3 adresses, ce n'est évidement plus le cas. On peut cependant affirmer que ces programmes ne respectent pas la définition du langage et qu'il est normal de devoir la respecter au moins en partie pour obtenir la vérification des bornes.

Un autre inconvénient relié au changement de la représentation des pointeurs est au niveau de la compatibilité. Si deux compilateurs utilisent les adresses comme représentation pour les pointeurs, le code généré par ces deux compilateurs peut être lié ensemble sans problèmes. Si les compilateurs utilisent une représentation différente, ça ne peut pas être fait directement. Il faut plutôt utiliser une couche d'adaptation qui converti les pointeurs d'un format à l'autre. On perd alors l'avantage des pointeurs bornés. De plus, on ne peut pas s'attendre à ce que le passage d'un pointeur de pointeur fonctionne correctement. Les bibliothèques existantes en format binaire seulement ne peuvent donc pas être réutilisées facilement. Le mieux est de les recompiler si leur code source est disponible.

L'utilisation de pointeurs bornés a aussi des effets sur la performance. Il faut s'y attendre puisqu'elle demande de manipuler des pointeurs au moins 3 fois plus gros et la vérification des bornes demande aussi un temps non négligeable. On peut calculer une perte de performance de l'ordre de 50% pour une implémentation bien optimisée de la vérification des bornes au moyen de pointeurs bornés et jusqu'à 200% dans le cas contraire.

Il existe une extension de GCC qui implémente cette approche. D'autres implémentations conservent encore plus d'informations dans le pointeur de manière à détecter toutes les erreurs dans l'utilisation des pointeurs, qu'elles soient spatiales ou temporelles [ABS93].

5.5.4.2 Représenter un pointeur par un descripteur

Si on ne veut pas modifier la taille des pointeurs, il est possible de les représenter par un descripteur qui est en fait un index dans une table de pointeurs. La table peut alors contenir des pointeurs bornés comme ils ont été définis à la section précédente.

Lorsqu'on modifie un pointeur, par exemple en l'incrémentant, sa représentation est inchangée, seulement son entrée dans la table est modifiée. Il ne peut pas y avoir deux pointeurs qui partagent le même descripteur. Si c'était le cas, le fait d'incrémenter un pointeur incrémenterait du même coup un autre pointeur, ce qu'on ne veut pas. Lorsqu'à un pointeur est copié, son entrée de la table des descripteurs est donc copiée vers celle du pointeur destination.

Cette approche comporte cependant un problème de taille. Le langage C permet de copier des pointeurs de bien des façons, et le compilateur ne peut pas toutes les «voir». Dans l'exemple 5.1, le compilateur ne peut pas permettre d'obtenir un résultat correct si les pointeurs sont représentés par des descripteurs.

Considérons le code C suivant (qui est correct).

```
int i[10];
int *p = i;
int *q;
memcpy(&q, &p, sizeof(int *));
p++;
```

Si les pointeurs sont représentés par des descripteurs, le résultat est incorrect puisque les deux pointeurs utilisent le même descripteur de pointeur et qu'ils sont tous deux incrémentés alors que seulement p devrait l'être.

Exemple 5.1: Problème de la représentation des pointeurs par des descripteurs

Pour la copie de pointeur, il faut donc interdire l'utilisation de `memcpy()` et toutes les autres fonctions qui copient de la mémoire. Le langage C++ a ce genre de restriction pour la copie des objets (`struct` ou `class`). La raison est que les objets peuvent avoir une sémantique de copie plus complexe définie au moyen de l'opérateur de copie. Les pointeurs représentés par des descripteurs ressemble donc plus à des des objets C++ qu'un type de données primitif.

Cette approche pour la vérification des bornes est habituellement rejetée parce qu'elle n'est pas compatible avec une fraction significative des programmes corrects.

5.5.4.3 Pistage des zones de mémoire

Une approche différente pour la vérification des bornes est possible sans modifier la représentation des pointeurs. Il faut alors conserver, lors de l'exécution, la définition des zones de mémoire allouée. Ces zones sont indexées de façon à ce qu'il soit rapide de retrouver à quelle zone appartient un pointeur. Une zone de mémoire peut être statique, allouée dynamiquement (`malloc()`) ou sur la pile. Elle correspond à une déclaration. Par exemple, si on déclare un tableau de structures qui contienne elles-mêmes des tableaux, il n'y a qu'une seule zone. Il n'est pas possible de diviser les zones plus finement sans risquer d'obtenir plusieurs zones pour une même adresse.

Chaque opération sur les pointeurs est modifiée pour vérifier qu'elle n'entraîne pas la sortie de la zone. Par exemple, avant d'incrémenter un pointeur, on vérifie dans quelle zone il se trouve et après l'incrémentation on vérifie que le pointeur n'a pas dépassé la fin de cette zone. Dès qu'un pointeur dépasse la zone à laquelle il appartient, une valeur spéciale lui est affectée. On appelle cette valeur `ILLEGAL` et elle est distincte du pointeur `NULL`. Cette valeur n'est pas strictement nécessaire, un programme pourrait être arrêté dès qu'il génère un pointeur «illégal». Il semble toutefois qu'un bon nombre de programmes génèrent des pointeurs de cette façon sans jamais les utiliser par la suite. La valeur `ILLEGAL` permet donc d'arrêter le programme seulement s'il utilise un tel pointeur.

Pour plusieurs des opérations qui demandent deux pointeurs, les deux doivent provenir de la même zone de mémoire, sinon l'opération n'a pas de sens. C'est le cas pour les opérateurs `-`, `<`, `<=`, `>`

et `>=`. Le compilateur ajoute donc cette vérification et le programme est arrêté si les deux pointeurs proviennent de zones différentes.

Le langage C permet d'obtenir un pointeur sur l'élément qui suit la fin d'un tableau même s'il ne permet pas de le déréférencer. Ceci permet par exemple d'utiliser de tels pointeurs dans la condition d'arrêt d'une boucle. Il n'est donc pas possible d'utiliser la valeur `ILLEGAL` pour représenter ces pointeurs. Pour éviter qu'il y ait ambiguïté entre le pointeur qui représente l'élément passé la fin d'un tableau et le premier élément du tableau suivant et pour conserver l'exactitude des vérifications, il est possible de conserver un minimum d'un octet entre les zones de mémoire.

Avant de déréférencer un pointeur, le compilateur vérifie toujours que la valeur n'est pas `ILLEGAL` et qu'elle n'a pas dépassé la limite de la zone à laquelle il appartient.

Le pistage des zones de mémoire demande donc des modifications à l'allocation de la mémoire à plusieurs endroits. Il y a deux aspects important à prendre en considération. Chaque zone de mémoire doit être ajoutée à la liste des zones lorsqu'elle est allouée, et elle doit être retirée lorsqu'elle est désaffectée. Il faut aussi s'assurer qu'il y a au moins un octet qui n'est pas alloué entre chaque zone. Ceci est facile à faire pour l'allocation dynamique en modifiant les fonctions `malloc()` et `free()`. Pour l'allocation statique et les variables, le compilateur doit être modifié et on n'a évidement pas à traiter la désaffectation. Pour les variables locales, le plus difficile est de gérer correctement la porté des déclarations. Elles doivent être ajoutées et retirer de la liste des zones correctement, même en présence de `goto`. Ceci est très semblable à ce qui se passe pour l'appel des constructeurs et des destructeurs en C++.

Pour les paramètres des fonctions, il est plus difficile de s'assurer qu'il y a un octet libre entre eux si on ne veut pas briser la convention d'appel. Heureusement, le cas le plus susceptible d'utiliser l'arithmétique des pointeurs sur un élément passé en paramètre se produit lors du passage par valeur d'une structure contenant un tableau. Ce cas est plutôt rare.

Si on veut traiter les paramètres correctement sans modifier les conventions d'appel, on peut effectuer une copie des paramètres dans les variables locales à l'entrée de la fonction et toujours utiliser cette copie. C'est le principe qui est utilisé dans [EY00] pour protéger les paramètres de certains débordements de tampon. Cette technique ne peut évidemment pas être appliquée lorsque le nombre de paramètres est variable.

Cette approche pour vérifier les bornes des tableaux a toutefois ses limites. En fait elle ne permet vraiment une vérification des bornes des tableaux, mais plutôt une vérification des bornes des zones. L'exemple 5.2 montre des cas où la vérification des bornes n'est pas effectuée correctement.

Cette approche réduit aussi considérablement la performance. Elle a été implémentée comme extension à GCC. Selon le programme qui est exécuté, le temps d'exécution peut être multiplié par un facteur pouvant aller jusqu'à 50.

5.5.4.4 Conserver une carte de la mémoire

Une autre approche pour vérifier les bornes des tableaux consiste à conserver une carte de la mémoire allouée. Elle est décrite dans [HJ91, Gin98, Jon95] À chaque octet de mémoire du programme, on fait correspondre un bit indiquant s'il est alloué ou non. Comme pour l'approche

```
// Le compilateur «voit» une zone pour s1
struct {
        char c[4];
} s1[10];

// et une autre zone pour s2
struct {
        char c[4];
        int i;
} s2;

int main()
{
        // Les accès hors bornes ne sont pas détectés
        // parce qu'ils sont à l'intérieur de la zone
        s1[0].c[20] = 1;
        s2.c[4] = 2;
        return 0;
}
```

Exemple 5.2: Le pistage des zones de mémoire ne permet pas toujours une vérification des bornes des tableaux

de la section 5.5.4.3, l'allocation de la mémoire doit être modifiée, au niveau du compilateur et de `malloc()`, pour tenir à jour la carte de la mémoire. On peut aussi conserver des blocs de mémoire libres entre les différentes zones pour mieux détecter les débordements. On les appelle zones rouges. En particulier, certaines parties de la pile comme les adresses de retour et les pointeurs de bloc de pile sauvegardés doivent être des zones rouges.

Il faut comprendre que cette approche est moins précise que celle de la section 5.5.4.3. Avec cette dernière, on a l'assurance que les pointeurs ne «sautent» pas d'une zone à l'autre. Ici, on vérifie seulement qu'ils pointent vers une adresse allouée. Chaque instruction qui effectue un accès à la mémoire est modifiée pour d'abord vérifier le bit indiquant si la mémoire est accessible ou non.

Cette approche ne permet pas uniquement la vérification des accès à des données non allouées, mais aussi à des données non initialisées. Pour ce faire, on conserve un bit supplémentaire pour chaque octet indiquant s'il est initialisé ou non. On se retrouve donc avec deux bits par octet, un pour indiquer s'il peut être lu et l'autre pour indiquer s'il peut être écrit. Valgrind implémente cette approche et est même en mesure de détecter l'utilisation de bits qui ne sont pas initialisés en conservant 8 bits supplémentaires par octet.

Il est aussi possible d'appliquer cette approche sur le code machine d'un programme lorsque le code source n'est pas disponible. Les outils Purify et Valgrind le permettent en modifiant ou en interprétant le code machine du programme qui doit être vérifier.

Cette approche a donc l'avantage de fonctionner avec pratiquement tous les programmes existant sans demander de modification, mais elle ne permet pas de détecter tous les débordements de tampon. Elle a aussi un impact significatif sur la performance des programmes. Purify ralenti l'exé-

cution des programmes par un facteur pouvant aller de 2 à 5. Ce type d'outil est donc généralement utilisé seulement pour le débogage.

5.6 Fonctionnalités du processeur

Chaque processeur possède des caractéristiques qui lui sont propres. Certaines d'entre elles peuvent être utilisées pour prévenir les débordements de tampons ou prévenir certaines conséquences de ces débordements.

5.6.1 Protéger les zones de mémoire sensibles

On a vu à la section 5.5.1.2 qu'il était possible d'utiliser certaines fonctionnalités des processeurs de la famille IA-32 pour protéger certaines parties de la mémoire contre l'écriture. En particulier, on cherchait à protéger les adresses de retour contre l'écriture. On peut généraliser ces techniques pour protéger d'autres parties de la mémoire. Cette section est basée sur [CPM+98, Woj98, Tec02, Cow00].

Si l'utilisation de registres de débogage pour cibler précisément une adresse à protéger est très spécifique aux processeurs IA-32, la protection au niveau des pages est permise par pratiquement tous les processeurs qui supportent la pagination. Pour éviter de constamment recourir au système d'exploitation, ce qui rendait pratiquement inutilisable la technique de la section 5.5.1.2, il faut allouer la mémoire en fonction des besoins de protection des différentes variables.

On peut s'organiser pour placer sur une même page seulement des données qui doivent être protéger. Lorsque les données sont initialisées, on demande au système d'exploitation de vérrouiller la page contre l'écriture. Si les données doivent être modifiées, on demande d'abord au système d'exploitation de dévérrouiller la page, on procède à la mise à jour de toutes les données qui doivent être modifiées, puis on demande à nouveau au système d'exploitation de vérrouiller la page. Les données sont donc protégées de toute modification involontaire de la part du programme, par exemple un débordement de tampon dans la page qui précède celle contenant les données sensibles. De cette façon on peut arrêter plusieurs types d'attaque.

La performance n'est pas trop impactée par cette technique car elle ne demande que deux interventions du système d'exploitation pour permettre la modification d'une quantité arbitraire de données. Il n'y a pas une intervention à chaque accès comme c'était le cas à la section 5.5.1.2.

Tout ce qu'il faut pour appliquer ce principe existe dans les systèmes d'exploitation modernes. Pas besoin d'adapter un compilateur. Par contre, l'application doit être modifiée pour indiquer ses «intentions». Pour cette raison, on applique généralement cette technique seulement aux données vraiment sensibles de l'application et qui ne sont pas modifiées souvent.

Voici une liste non exhaustive d'objets qui peuvent être intéressants à protéger. La protection de ceux qui ne sont pas sous le contrôle de l'application demanderait des modifications au compilateur.

- Variables de sécurité (authentification, description des droits, ...)
- Pointeurs de l'application

– Adresses de retour
– Pointeurs de bloc de pile sauvegardé
– Global Offset Table (GOT)
– Table des pointeurs de fonctions virtuelles
– Table des pointeurs de la fonction `atexit()`

Un exemple d'utilisation de cette fonctionnalité se trouve dans les applications compilées avec StackGuard. Ce dernier protège ainsi la table des canaris du programme [BK00, Con99]. Cette table est générée de façon aléatoire au début du programme et elle est par la suite utilisée pour protéger les adresses de retour (section 5.5.1.1). Si un attaquant pouvait écraser la table de canaris, il lui serait très facile de déjouer la protection de StackGuard. C'est pour cette raison que la table ne doit pas pouvoir être modifiée après le démarrage du programme.

Aussi, le code des applications est généralement protégé contre l'écriture par défaut sur la plupart des systèmes d'exploitation. Il en est ainsi pour permettre plus facilement le partage du code entre différentes applications qui utilise, par exemple, une même bibliothèque. De plus, si le système d'exploitation manque de mémoire physique, il peut sans problème recycler une page de code non modifiée sans avoir à l'écrire dans le fichier d'échange puisqu'il peut la recharger directement du fichier d'origine.

Bien que la protection des zones de mémoire sensibles ait le potentiel de réduire considérablement les conséquences des débordements de tampons, elle ne les empêche pas directement. De plus, même si on pouvait protéger efficacement toutes les données sensibles d'un programme, il reste qu'il serait très difficile d'avoir l'assurance que toutes ces données sont bien identifiées. Leur identification demanderait une connaissance approfondie du programme en question, mais aussi du système d'exploitation, du compilateur, de son environnement d'exécution ainsi que de toutes autres bibliothèques qui seraient utilisées.

5.6.2 Pile non exécutable

Cette section est basée sur [Des97b, Des02, Des97a, Woj98].

Nous avons déjà mentionné, entre autres à la section 3.6.1, qu'il était possible d'avoir une pile non exécutable. Ceci permet d'empêcher les attaques où le code à exécuter se trouve sur la pile. Il faut dire que la pile est la zone de mémoire la plus souvent utilisée par les attaquants. Sur les systèmes d'exploitation de type Unix, les variables d'environnement, les arguments des programmes ainsi que tous les tampons non statiques déclarés localement se trouvent sur la pile.

La retouche du Openwall Project pour Linux qui empêche l'exécution de code sur la pile est très spécifique aux processeurs de la famille IA-32. Elle fonctionne en brisant la symétrie dans la définition des segments. Normalement, sous Linux, les segments de code, de données et de la pile (CS, DS et SS) permettent tous d'accéder à la même mémoire. Cette retouche modifie le segment de code de façon à ne pas couvrir la pile qui se trouve à la fin de l'espace d'adressage.

L'utilisation d'une pile non exécutable comporte cependant des inconvénients. Certains programmes légitimes placent du code qui doit être exécuté sur la pile. Par exemple, lorsqu'on utilise une extension du compilateur GCC permettant la définition de fonctions imbriquées en C, le compilateur doit générer des «trampolines» pour permettre l'utilisation de pointeurs vers ces fonctions

imbriquées. Les trampolines sont du code permettant aux fonctions imbriquées de retrouver leur contexte d'exécution (variables locales de la fonction externe). Lorsqu'un pointeur sur une fonction imbriquée est généré, le trampoline, et donc le code, est placé sur la pile. Ce mécanisme ne fonctionne plus lorsque la pile est non exécutable.

La retouche offre la possibilité de détecter et d'émuler les trampolines. Elle effectue cette détection en vérifiant que l'instruction qui transfère le contrôle à la pile est un `call` plutôt qu'un `ret`. Si c'est le cas, le `call` est émulé. Ensuite, un nombre très limité d'instructions peuvent être émulées dans le trampoline.

Un autre problème posé par la pile non exécutable se produit lors du traitement des signaux. Lorsqu'un signal est envoyé à un processus, Linux place le code permettant le retour au programme principal sur la pile. Ceci ne fonctionne plus lorsque la pile est non exécutable. Pour régler ce problème, la méthode utilisée pour retourner d'un signal a été modifiée, mais cette modification n'est pas visible par les applications, seulement par le noyau.

Il existe d'autres programmes qui utilisent la pile pour y placer du code. Par exemple, les programmes écrits dans des langages fonctionnels sont susceptibles d'utiliser la pile pour l'exécution de code. LISP et Objective C en sont des exemples. Il faut donc prévoir un mécanisme pour désactiver la protection contre l'exécution de code sur la pile si on veut que ces programmes continuent de fonctionner.

Quoi qu'il en soit, empêcher l'exécution de code sur la pile n'empêche aucunement les débordements de tampons. Dans le meilleur des cas, ceci permet d'empêcher certaines attaques de réussir, mais il existe plusieurs moyens pour déjouer cette protection [Woj98, BK00, Woj01].

5.6.3 Autres données non exécutables

Cette section est basée sur [Tea00, Woj01, Dup02, Tea02, Sta01, Con99].

Rendre non exécutables les données qui se trouvent à l'extérieur de la pile est un peu plus difficile. La pile se trouve à la fin de l'espace d'adressage du programme, il était donc possible de simplement diminuer la taille du segment de code. Par contre, les zones de données se trouvent en général intercalées entre des zones de code. Ceci ne poserait pas de problème si le processeur permettait de définir pour chaque page si elle est exécutable ou non, mais ce n'est pas le cas pour les processeurs de la famille IA-32.

Quoi qu'il en soit, deux techniques permettent d'obtenir ce résultat. La première consiste à exploiter une particularité du fonctionnement interne du processeur. Tous les processeurs qui supportent la pagination de la mémoire conservent un certain nombre d'entrées de pages dans une mémoire cache pour éviter de rechercher la page en mémoire principale à chaque accès à la page. On appelle cette mémoire cache répertoire des pages actives ou TLB[4]. Si l'entrée de la table est modifiée en mémoire principale, l'entré du TLB reste intacte.

Pour tous les processeurs de la famille IA-32 et leurs clones, les processeurs ont des TLB séparés pour le code (ITLB) et les données (DTLB). Bien que ces deux TLB utilisent la même source pour les entrées des pages, il est possible d'avoir un d'entre eux chargé et pas l'autre. Cette particularité a été

[4]De l'anglais : *translation look-aside buffer*.

découverte par le projet plex86[5] qui l'utilise pour virtualiser le processeur sans que les applications ne puissent s'en apercevoir. Le noyau du système d'exploitation peut de plus intercepter le chargement du ITLB et du DTLB en marquant une page non présente ou non accessible par l'utilisateur. Il peut donc contrôler précisément quelles pages se retrouvent dans le ITLB et donc quelles pages peuvent être exécutées.

Cette technique a été implémentée par PaX, une extension de Linux. Elle ralentie un peu l'exécution des programmes qui utilisent beaucoup de pages à la fois, mais en général la perte de performance est seulement de 5% à 8%.

La deuxième technique utilise le mécanisme de segmentation plutôt que celui de la pagination. C'est la méthode «normale» d'empêcher l'exécution sur les processeurs IA-32. Le problème est que les programmes pour Linux ne manipulent pas les segments en général. Ils utilisent un modèle de mémoire où tout l'espace d'adressage, généralement 3 gigaoctets de mémoire virtuelle, est accessible par chacun des segments sans restriction. Lorsqu'on utilise la segmentation pour empêcher l'exécution des données, il est nécessaire de maintenir la symétrie dans la définition des segments de code et de données pour que les programmes existants puissent s'exécuter correctement, c'est-à-dire que les pages de mémoire doivent être accessibles au même déplacement dans les deux segments. Chaque page doit donc être placée à deux endroits dans l'espace d'adressage linéaire puisqu'elle peut être accessible en tant que données, mais pas en tant que code. Ceci divise par deux la mémoire virtuelle disponible pour l'application. Les programmes qui utilisent de gros fichiers mappés peuvent donc souffrir de cette technique.

Puisque cette technique utilise des «vrais» fonctionnalités du processeur pour accomplir son travail, on peut s'attendre à ce que la performance soit meilleure. La perte de performance est en effet presque nulle. Cette technique est implémentée par kNoX et RSX.

De la même façon que pour la pile, empêcher l'exécution de code dans les autres données n'empêche aucunement les débordements de tampons. Ceci bloque certaines attaques, mais il est possible de passer outre cette protection [Woj01].

5.6.4 Pile qui grandit vers le haut

On a vu qu'avec la plupart des systèmes d'exploitation et la plupart des processeurs, la pile grandit vers le bas (adresses basses). Lorsque la pile grandit vers le haut, il est impossible d'exploiter le débordement d'un tampon qui se trouve au sommet de la pile puisque aucune donnée ne se trouve après le tampon et que les débordements se font généralement à la fin d'un tampon. Il existe des processeurs où la pile grandit nativement vers le haut ou qui supportent une pile qui peut grandir dans un sens ou l'autre. Par exemple, les ordinateurs VAX supportaient une pile qui grandissait vers les adresses hautes [KS02].

Même si la pile grandit vers le haut, il est possible d'exploiter le débordement d'un tampon qui n'est pas au sommet de la pile. En particulier, un tampon défini localement dans une fonction appelante permet d'écraser une adresse de retour s'il déborde. La figure 5.1 montre ce qui se passe.

Bien qu'une pile grandissant vers le haut puisse empêcher d'exploiter certains cas bien précis de débordement, son efficacité est très limitée. De plus, supporter une pile qui grandit vers le haut sur

[5] www.plex86.org

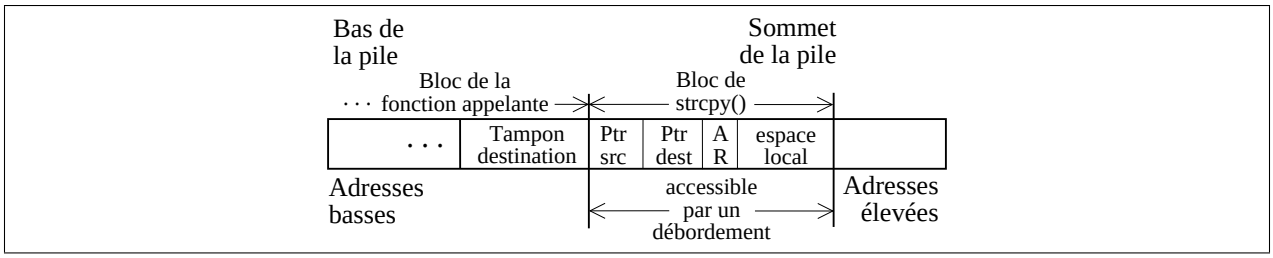

Figure 5.1: Débordement sur une pile qui grandit vers le haut. Une fonction a appelé `strcpy()` en lui passant un tampon local. S'il y a un débordement, l'adresse de retour peut être écrasée comme on l'a vu à la section 3.5.1. Il faut remarquer que l'attaquant peut prendre le contrôle plus «rapidement» que sur une pile qui grandit vers le bas puisque le retour s'effectue dès la fin de `strcpy()`.

un processeur qui ne le permet pas nativement serait très peu performant. Pour ces raisons, cette approche n'est habituellement pas utilisée.

5.6.5 Segmentation

Cette section est basée sur [Jon95, Int00, KS02].

Les processeurs Intel de la famille IA-32 offrent un mécanisme de segmentation qui permet un découpage fin des données. Linux utilise un modèle de mémoire où l'application n'a pas besoin de se soucier des segments. En particulier, les segments de code et de données couvrent généralement la totalité de l'espace d'adressage. Il serait toutefois possible de définir un segment pour chaque zone de mémoire. De cette façon, on pourrait obtenir des résultats très semblables à ceux de la section 5.5.4.3 mais avec de bien meilleures performances grâce à l'aide du processeur.

Définir les zones de mémoire à l'aide de segments demande des modifications au compilateur. Le système d'exploitation doit aussi intervenir puisque la modification des descripteurs de segment est une opération privilégiée. Contrairement au pistage des zones de mémoire, la segmentation ne demande pas de laisser de l'espace entre les blocs de mémoire. Elle oblige cependant à modifier la représentation des pointeurs pour inclure un sélecteur de segment de 16 bits, 32 bits ne suffisent donc plus pour représenter un pointeur. Un autre inconvénient de cette approche est qu'elle ne permet pas d'utiliser plus de 16 384 segments à la fois. Pour les programmes qui manipulent un grand nombre de données, ce n'est probablement pas suffisant. Il faudrait donc prévoir un descripteur global pour les zones qui ne peuvent pas être représentées par leur propre descripteur et peut-être aussi allouer «intelligemment» les descripteurs disponibles.

Si cette technique ne permet pas de détecter tous les débordements de tampons, elle permet au moins de les contenir à l'intérieur de leur zone. Ceci est beaucoup mieux que la plupart des autres techniques mentionnées.

5.6.6 Empêcher les accès non alignés

Certains processeurs obligent les programmes à effectuer des accès alignés en mémoire, c'est-à-dire à une adresse qui est un multiple de la taille du mot qui est lu ou écrit en mémoire. Les

processeurs de la famille IA-32 ne l'obligent pas par défaut, mais ils peuvent être configurés pour l'obliger. Effectuer des accès à la mémoire qui sont toujours alignés permet à un programme de s'exécuter plus rapidement, cette fonctionnalité permet donc de découvrir des accès à la mémoire qui pourraient être optimisés. Elle est décrite dans [Int00].

Bien que cette fonctionnalité soit là d'abord pour des raisons de performance, elle peut être utilisée pour contrer certaines techniques utilisées par les pirates. Par exemple, nous avons vu à la section 4.2.2 qu'une des techniques permettant d'écraser un mot avec une valeur arbitraire dans une vulnérabilité de chaîne de format passait par l'utilisation d'adresses non alignées. Empêcher l'écriture à des adresses non alignées peut donc empêcher un certain nombre d'attaques.

5.7 Autres techniques dynamiques

Cette section décrit d'autres techniques dynamiques qui n'empêchent pas nécessairement les débordements de tampons ni les vulnérabilités de chaîne de format, mais qui permettent d'éviter ou de rendre plus difficiles certaines formes communes d'exploitation.

5.7.1 Utiliser des bibliothèques alternatives

Les bibliothèques utilisées par défaut sur les systèmes, par exemple la bibliothèque C, sont habituellement conçues pour respecter certaines spécifications et s'exécuter le plus rapidement possible. Il est parfois possible d'utiliser des versions alternatives des bibliothèques qui effectuent plus de vérifications que les spécifications demandent de manière à détecter des mauvais comportements de programmes. En plus de vérifications supplémentaires, il est parfois possible d'effectuer le travaille de façon différente pour détecter plus facilement des mauvais comportements.

5.7.1.1 Remplacer malloc() et ses amis

Cette section est basée sur [Per93].

Il est possible de remplacer les fonctions qui gèrent l'allocation de la mémoire pour aider la détection des débordements de tampons. Le principe est assez simple. La mémoire est allouée de manière à coïncider avec le début ou la fin d'une page de mémoire. Lorsque la mémoire est allouée à la fin de la page, l'allocation de la mémoire marque la page suivante comme inaccessible. De cette façon, le processeur signale une faute dès que le programme dépasse la capacité de la zone de mémoire allouée. Lors de la détection de l'erreur, un débogueur peut montrer l'instruction exacte qui cause le débordement. Si la mémoire est allouée au début d'une page, c'est la page précédente qui est marquée comme inaccessible et ce sont les débordements vers le bas qui sont détectés.

Il peut y avoir des problèmes d'alignement dans certains cas si la quantité de mémoire allouée n'est pas un multiple de la taille des mots native du processeur. En effet, les programmes s'attendent à recevoir un bloc de mémoire aligné lors de l'allocation et la position des données à l'intérieur du bloc est calculée en fonction de cet alignement. Heureusement, les compilateurs ajoutent habituellement des octets de remplissage aux structures de manière à ce qu'elles aient une taille multiple de celle d'un mot. Ceci fait en sorte que la mémoire se trouve habituellement alignée.

Cependant, si un programme réservait certain nombre d'octets qui n'est pas un multiple de la taille d'un mot, la mémoire ne serait pas alignée et il pourrait y avoir un problème. Pour contrer cet effet, la fonction d'allocation peut choisir d'allouer quelques octets de remplissage supplémentaires. Par exemple, si un programme demande 5 octets et que la taille d'un mot est de 4 octets (32 bits) ce sont alors 8 octets qui sont alloués. De cette façon il n'y a plus de problèmes d'alignement, mais l'inconvénient est que les débordements sont détectés seulement s'ils dépassent 3 octets.

Cette technique ne permet pas de détecter tous les débordements. Par exemple, si le bloc de mémoire alloué correspond à une structure qui contient un ou des tableaux, ces tableaux peuvent déborder sans qu'il y ait débordement du bloc de mémoire alloué, et donc sans détection. Ces limitations sont très similaires à celles des techniques présentées aux sections 5.5.4.3, 5.5.4.4 et 5.6.5. La différence est qu'ici il faut choisir entre détecter les débordements après la fin des blocs de mémoire ou celles avant le début. Un autre inconvénient de cette technique est qu'elle demande beaucoup de mémoire. Même si seulement quelques octets sont demandés par le programme, ce sont deux pages complètes de mémoire qui sont requises, une pour les données, l'autre comme garde.

Cette technique est implantée dans la bibliothèque Electric Fence.

5.7.1.2 Remplacer gets() et ses amis

Cette section est basée sur [BTS99, Ava, TS01, Sna97, Sna00, Whe02c, One].

On couvre ici un grand nombre de fonctions de la bibliothèque C. Le but est d'éviter le pire pour les fonctions qui ne permettent pas de spécifier la taille du tampon passé. Parmi ces fonctions, on retrouve `gets()`, `strcpy()`, `sprintf()`, `sscanf()` et leurs dérivés.

Plus précisément, on cherche à protéger les adresses de retour et les pointeurs de bloc de pile sauvegardé contre l'écrasement. Nous avons vu au chapitre 3 que ces adresses sont des cibles de choix pour les attaques exploitant les débordements de tampons. Les pointeurs de bloc de pile forment une chaîne qui permet d'identifier la position des adresses de retour et des pointeurs de bloc de pile sauvegardés. Ils marquent la limite entre les variables locales et les paramètres d'une fonction. Les tableaux ne peuvent pas passer par dessus ces valeurs. Il est donc possible au moment de l'exécution de vérifier qu'on ne traverse pas une de ces limites lors d'une opération qui rempli un tampon. À défaut d'éviter les débordements, on évite plusieurs formes d'exploitation.

Cette technique a été implémentée sous Linux dans une bibliothèque appelée libsafe qui peut être liée dynamiquement aux programmes. Les programmes n'ont donc pas à être recompilés pour bénéficier de la protection de cette bibliothèque. Sous FreeBSD, c'est la bibliothèque libparanoia qui offre une protection similaire.

La technique pourrait aussi s'appliquer aux fonctions auxquelles on passe la taille du tampon pour valider que la taille du tampon est «vraisemblable», mais en général on se fie que si un programme prend la peine de donner la taille d'un tampon, la taille qu'il donne est la bonne.

Cette technique a cependant ses limites. D'abord, elle ne détecte pas vraiment les débordements de façon général, elle s'attarde plutôt à protéger certaines valeurs sur la pile. On a vu au chapitre 3 qu'il existe d'autres valeurs qui peuvent être la cible d'attaques. En outre, cette technique n'offre aucune protection si un programme est compilé sans les pointeurs de bloc de pile

(`-fomit-frame-pointer`). Elle ne peut évidement rien contre les débordements qui se produisent sur le tas. Aussi, les bibliothèques de remplacement ne sont pas compatibles avec certains compilateurs qui altèrent le format du bloc de pile, comme c'est le cas avec StackGuard.

5.7.1.3 Remplacer printf() et ses amis

Cette section est basée sur [TS01, Rob01].

On peut vouloir se protéger des vulnérabilités de chaîne de format par le remplacement des fonctions qui traitent des chaînes de format. On s'intéresse plus particulièrement au spécificateur de conversion «`%n`» qui permet d'écrire des valeurs arbitraires en mémoire. Pour ce faire il existe deux approches.

On peut d'abord utiliser le même principe que celui expliqué à la section précédente pour empêcher l'écrasement des adresses de retour et des pointeurs de bloc de pile. L'adresse de chaque «`%n`» est alors vérifiée avant d'être utilisée. Il est de plus possible de vérifier que les arguments variables qui sont convertis un après l'autre ne passent pas par dessus un autre pointeur de bloc de pile. Cette situation est impossible si le bon nombre d'argument a été passé à la fonction qui prend une chaîne de format. Ces vérifications ont été implémentées dans `libsafe 2.0`. Elle a l'avantage de ne donner aucun faux positif puisque aucun programme C correct ne peut modifier son adresse de retour ou un pointeur de bloc de pile. Le problème est qu'elle n'empêche aucunement d'écraser d'autres valeurs et on a vu au chapitre 4 qu'il y avait plusieurs autres cibles.

Une autre approche est d'empêcher complètement l'utilisation du spécificateur de conversion «`%n`». Cette approche est beaucoup plus drastique car elle empêche même les utilisations légitimes de ce spécificateur de conversion, comme le calcul de la quantité de mémoire nécessaire pour construire une chaîne de caractères contenant certaines variables. Par contre, son utilisation est assez rare et plusieurs programmes peuvent fonctionner correctement même en utilisant une bibliothèque qui l'empêche.

5.7.1.4 Utiliser des macros pour compter les paramètres

Cette section est basée sur [CBB+01].

Il existe une autre approche offrant une meilleure précision pour la détection des usages abusifs des spécificateurs de conversion. Le principe est simple. Les fonctions qui prennent en paramètre une chaîne de format sont remplacées par des macros. Ces macros comptent le nombre de paramètres qui sont réellement passés au moment de la compilation et passent cette information à une version modifiée de la fonction qui accepte un paramètre supplémentaire. Lors de l'exécution, la fonction modifiée s'assure de ne pas convertir plus de paramètres que le nombre qui a été réellement passé.

L'inconvénient de cette technique est qu'elle demande une recompilation des programmes à protéger. Par contre, le code source n'a pas à être modifié puisque la définition des macros se trouve dans les fichiers d'entête normalisés. FormatGuard implémente cette approche par la modification de glibc.

5.7.1.5 Utiliser des interfaces différentes

Cette section est basée sur [Whe02c, MdR99, Mil00].

On a vu aux sections 2.3 et 2.4 que certaines fonctions normalisées de la bibliothèque C ne permettent pas facilement le respect des bornes des tampons, et que d'autres avaient des interfaces incohérentes qui nuisaient à leur utilisation correcte. Il existe des fonctions avec des interfaces alternatives qui permettent d'écrire plus facilement du code sans débordement.

strlcpy() et strlcat() Les fonctions `strlcpy()` et `strlcat()` ne sont pas normalisées, mais elles offrent une alternative de plus en plus populaire aux fonctions de la bibliothèque C. Elles ont été créées par le projet OpenBSD qui a pour objectif premier la sécurité de son système d'exploitation. Ces fonctions sont cohérentes dans le sens qu'on leur fournit toujours la taille totale du tampon, elles terminent toujours la chaîne de caractères par un caractère nul et elles retournent toujours la longueur total de la chaîne qu'elles auraient créé s'il n'y avait pas eu de troncation. Sur les systèmes Linux, elles sont le plus souvent disponibles dans la bibliothèque glib sous les noms `g_strlcpy()` et `g_strlcat()`.

astring de libmib La bibliothèque libmib contient une partie appelée astring qui offre une alternative à certaines fonctions de la bibliothèque C. Le principe est assez simple. Plutôt que de passer en paramètre un pointeur à un tampon, on passe plutôt un pointeur à un pointeur de tampon. De cette façon, la bibliothèque peut allouer un tampon de la bonne taille et donner au programme un pointeur vers ce tampon. Cette bibliothèque est décrite et distribuée à [Cav].

5.7.2 Modifier l'emplacement des bibliothèques

Plusieurs attaques exploitent le fait que les bibliothèques sont toujours chargées au même endroit en mémoire. Par exemple, la bibliothèque C contient à peu près tout le code qu'un attaquant peut désirer exécuter. Ces attaques ne peuvent fonctionner que si l'attaquant est en mesure de détourner le cours d'exécution vers l'endroit exact où le code qui lui est utile se trouve. Avec Linux, il est possible de charger les bibliothèques partagées à n'importe quelle adresse puisqu'elles sont composées de code indépendant de la position.

Cette section est basée sur [Des97a, Woj98, Woj01].

5.7.2.1 Octet nul dans l'adresse

La présence d'un octet nul dans l'adresse des bibliothèques rend plus difficiles les attaques qui utilisent le code déjà présent dans les bibliothèques, comme l'attaque décrite à la section 3.6.1. En effet, pour ce genre d'attaque, il faut placer sur la pile l'adresse de la fonction qui doit être appelée suivi des paramètres de la fonction. Si l'adresse de retour contient un caractère nul et que la chaîne est copiée au moyen d'une fonction comme `strcpy()`, la copie s'arrête dès que le caractère nul est rencontré et il n'est pas possible de fournir des paramètres arbitraires à la fonction.

Cette mesure de protection a été implémentée pour la première fois dans la retouche du Openwall Project empêchant l'exécution de code sur la pile. Elle a été reprise par la retouche kNoX qui empêche aussi l'exécution des autres données modifiables. Le programme principal ne peut pas être protégé de cette façon car il n'est pas composé de code indépendant de la position.

5.7.2.2 Adresse aléatoire

Une autre façon de se protéger contre ces attaques est de charger le code des bibliothèques à une adresse aléatoire. Cette approche est implémentée par PaX. Comme pour le cas des adresses avec un octet nul, le programme principal ne peut pas être déplacé. Il existe aussi certains moyens pour un attaquant de découvrir lors de l'exécution l'adresse d'une bibliothèque [Woj01].

5.7.3 Modifier l'emplacement de la pile

Plusieurs attaques ont besoin de l'adresse exacte ou approximative de variables ou de tampons sur la pile. En choisissant de façon aléatoire l'endroit où la pile débute, il est possible de contrer plusieurs de ces attaques. Cette approche est expliquée dans [Ket98] avec un exemple montrant que cette mesure de protection peut être intégrée directement dans les programmes. Pax implémente cette mesure au niveau du noyau de Linux, ce qui permet d'appliquer la méthode à tous les programmes sans modification ni recompilation [Woj01].

Cette approche n'est cependant pas à toutes épreuves car un attaquant peut essayer plusieurs possibilités jusqu'à ce qu'il trouve la bonne. D'autres types d'attaques ne demandent pas de connaître l'adresse de variables sur la pile.

5.7.4 Vérifier que l'exécution correspond à un modèle

La technique présentée dans cette section ne détecte pas du tout les débordements de tampons ni les vulnérabilités de chaîne de format, elle détecte plutôt l'exploitation de ces vulnérabilités. Elle effectue en quelque sorte de la détection d'intrusions. Elle est décrite en détail dans [Wag00].

Le principe se base sur le fait qu'un pirate ne peut pas effectuer d'actions malveillantes sur un système sans faire appel au noyau du système d'exploitation. Tout au plus, il pourrait effectuer un déni de service. En effet, que ce soit pour la lecture ou l'écriture d'un fichier, l'établissement ou l'utilisation d'une connexion réseau ou l'exécution d'un programme, le noyau doit intervenir. De plus, il est possible lors de l'exécution de suivre la trace des appels au noyau effectués par le programme.

La protection d'un programme à l'aide de cette technique demande deux grandes phases. La première consiste à construire un modèle du programme. Ce modèle doit représenter les appels au noyau que le programme peut effectuer. La deuxième phase se déroule lors de l'exécution du programme. On vérifie alors en temps réel si l'exécution du programme correspond au modèle.

Les modèles peuvent être construits de différentes façons. Plutôt que de les construire manuellement, de façon empirique ou à partir de spécifications externes, David Wagner propose de construire

les modèles à partir du code source du programme. Il suppose donc que le programme est écrit avec de bonnes intentions.

Le modèle peut s'exprimer façons très diversifiées. Il peut représenter le programme de façon très grossière, très précise ou n'importe quoi entre les deux.

5.7.4.1 Un modèle trivial

Un modèle très simple d'un programme pourrait être constitué de l'ensemble de tous les appels au noyau que le programme peut effectuer. Ce modèle a l'avantage d'être facile à vérifier. À chaque appel au noyau effectué lors d'une exécution du programme, on vérifie si cet appel fait partie de l'ensemble des appels au noyau autorisés. Si ce n'est pas le cas, on peut arrêter le programme immédiatement. Par contre, ce type de modèle n'est pas très précis car il ne tient aucunement compte de l'ordre des appels.

5.7.4.2 Automate fini

Un modèle plus précis peut être construit en considérant le graphe de flot de contrôle du programme. Chaque noeud de ce graphe doit effectuer au plus un appel au noyau. Les appels de procédures sont représentés par deux noeuds qui ne sont pas reliés directement ensemble. Le premier est relié au point d'entée de la procédure appelé et le point de sortie de cette procédure est relié au second. Le graphe de flot de contrôle est $G = \langle V, E \rangle$.

On crée un automate fini non déterministe à partir de ce graphe de la façon suivante.

- L'ensemble des états est $V \cup \{\mathsf{Wrong}\}$.
- L'alphabet Σ est l'ensemble des appels au noyau.
- Soit $(v, w) \in E$, il y a une transition $v \xrightarrow{a} w$ s'il y a un appel $a \in \Sigma$ effectué par le noeud v, sinon il y a une transition $v \xrightarrow{\lambda} w$. λ représente la chaîne vide.
- Tous les états sont accepteurs sauf Wrong.

Lors de l'exécution du programme, on vérifie si la séquence des appels au noyau que le programme effectue est acceptée par l'automate. Pour effectuer cette vérification, on n'a pas besoin d'attendre de connaître la séquence complète. On peut simuler l'automate non déterministe à mesure que les appels au noyau sont effectués. Si on définit S_s comme l'ensemble des états qui peuvent être atteints après avoir lu la séquence s, on peut conclure que la séquence sera rejetée par l'automate peu importe ce qui suit dès que S_s ne compte plus d'état accepteur puisque aucune transition n'a été définie à partir d'un état non accepteur.

Ce type de modèle est sain dans le sens qu'il ne provoque aucune fausse alarme. L'automate est construit de manière à couvrir tous les chemins d'exécution possibles du programme. Par contre, il couvre aussi plusieurs chemins d'exécution impossibles. Il permet en effet que le retour de l'appel d'une procédure se fasse à un noeud qui n'est pas celui correspondant à celui qui a effectué l'appel de la procédure. Une attaque qui suivrait un de ces chemins serait donc non détectée.

5.7.4.3 Automate à pile

Il est possible d'utiliser un automate à pile non déterministe pour obtenir un modèle encore plus précis. Considérons le même graphe de flot de contrôle $G = \langle V, E \rangle$ qu'à la section précédente. Le noeud d'entrée d'une fonction f est noté Entry(f) et le noeud v' est celui auquel on retourne après un appel initié par le noeud v. L'automate à pile est définit de la façon suivante.

- L'alphabet d'entrée Σ est l'ensemble des appels au noyau.
- L'alphabet de la pile est $V \cup \Sigma$.
- Les transitions sont :
  - Si $v \in V$ est un noeud d'appel de fonction au sommet de la pile, on le dépile, on empile v', puis Entry(f).
  - Si $v \in V$ est un noeud de sortie de fonction au sommet de la pile, on le dépile.
  - Si $v \in V$ est une autre sorte de noeud au sommet de la pile et qu'il effectue un appel au noyau, on le dépile, on empile le successeur w tel que $(v, w) \in E$ puis on empile $s \in \Sigma$ l'appel au noyau (non déterministe).
  - Si $v \in V$ est une autre sorte de noeud au sommet de la pile et qu'il n'effectue pas d'appel au noyau, on le dépile puis on empile le successeur w tel que $(v, w) \in E$ (non déterministe).
  - Si $s \in \Sigma$ est au sommet de la pile et que s est le prochain symbole d'entrée, on lit s et on dépile s.
  - Si $s \in \Sigma$ est au sommet de la pile et que s' est le prochain symbole d'entrée et que $s \neq s'$, on lit s' et on entre dans l'état Wrong.

On peut ici aussi rejeter la séquence dès que l'ensemble des états qu'on peut atteindre après avoir lu une certaine séquence ne contient que l'état Wrong. Cet automate à pile a l'avantage de faire toujours correspondre le retour de l'appel d'une fonction au véritable noeud appelant. Bien qu'il permette d'explorer des chemins qui ne correspondent pas à l'exécution du programme, on est certain, que le chemin suivit par une exécution correcte du programme sera exploré par l'automate. Il ne provoque donc pas de fausse alarme lui non plus.

Le modèle, bien que plus précis que celui construit au moyen d'un automate fini, n'est pas parfait dans le sens qu'il ne permet pas de détecter toutes les intrusion. Un attaquant minutieux pourrait toujours construire une attaque qui effectue des appels au noyau dans un ordre permis par l'automate.

Un problème important de ce modèle est comment simuler efficacement l'automate à pile non déterministe. On peut bien sûr construire une grammaire non contextuelle qui correspond à l'automate à pile, mais l'analyse syntaxique pose problème. Pour être en mesure de vérifier si une séquence partielle d'appels au noyau correspond au début d'une séquence admise par la grammaire tout en tenant compte de la pile du programme, il semble qu'une analyse descendante soit plus appropriée qu'une analyse ascendante. Malheureusement, la plupart des méthodes connues permettant de reconnaître un langage non contextuel arbitraire effectuent une analyse ascendante.

David Wagner propose un algorithme descendant permettant de reconnaître une grammaire non contextuelle et ayant les caractéristiques suivantes.

- Il permet une analyse «en ligne», c'est-à-dire qu'il est en mesure de déterminer si une séquence partielle correspond au début d'au moins une séquence appartenant au langage.

– Il est relativement efficace. Son temps d'exécution est cubique en fonction de la longueur de la séquence dans le pire des cas, ce qui est beaucoup mieux que le temps d'exécution exponentiel obtenu par la simulation «naïve» de l'automate à pile non déterministe.
– Il permet un accès en temps réel à l'ensemble des arbres syntaxiques possibles pour un début de séquence donnée.

Cette dernière caractéristique est nécessaire pour contourner certains problèmes que nous allons voir à la section 5.7.4.5.

Cet algorithme utilise le fait que les configurations de pile atteignables suite à la lecture d'une séquence partielle par un automate à pile non déterministe forment un langage régulier. Ce langage régulier est représenté au moyen d'un automate fini non déterministe. Après la lecture de chaque symbole de la séquence, l'algorithme permet de modifier cet automate pour qu'il décrive les nouvelles configurations de pile possibles.

Selon le programme qui est surveillé au moyen de cet algorithme, le temps d'exécution peut être acceptable ou pas. Pour certains programmes surveillés, le temps d'exécution peut être augmenté de plusieurs minutes voire plus d'une heure pour un seul événement interactif. Pour ces programmes, il faut trouver un modèle qui permet une vérification plus efficace.

5.7.4.4 Graphe orienté

Un autre modèle possible peut être construit en considérant, parmi toutes les séquences possibles d'appels au noyau, les fenêtres de longueur k. Un graphe orienté peut donc être construit avec des noeuds représentés par des séquences de longueur $k-1$ et des arrêtes $(s_1 s_2 \dots s_{k-1}, s_2 s_3 \dots s_k)$ qui indiquent que le programme possède au moins une trace d'exécution qui accepte la séquence $s_1 s_2 \dots s_k$.

La surveillance au moyen d'un graphe orienté est très efficace, mais le problème réside en la construction de ce graphe. La technique utilisée par David Wagner est si peu efficace qu'elle n'a pas permis d'expérimenter avec des séquences plus longues que 2 appels au noyau. Quoi qu'il en soit, ce modèle peut être très intéressant, surtout si on parvient à augmenter la longueur de la fenêtre.

5.7.4.5 Flot de contrôle non standard

Le graphe de flot de contrôle d'un programme C n'est généralement pas suffisant pour identifier tous les chemins d'exécution possibles. Par exemple, il ne représente aucunement les signaux qui sont traités de manière asynchrones par un programme Unix. Heureusement, le même mécanisme qui permet d'intercepter les appels au noyau permet aussi d'intercepter les signaux. À ce moment on peut suspendre la surveillance du programme et la reprendre au moment où le traitement du signal se termine. Le traitement du signal peut être lui aussi surveillé si on a pris soin de prendre en note la fonction qui doit traiter le signal.

Les modèles n'ont donc pas à être modifiés pour la gestion des signaux, seul le programme de surveillance a besoin d'en tenir compte.

Un autre problème se pose lors de l'utilisation des fonctions `setjmp()` et `longjmp()`. Ces fonctions permettent respectivement de sauvegarder une partie du contexte d'exécution (pointeur d'instruction, pointeur de pile) et d'y revenir plus tard, souvent suite à la détection d'une erreur. Puisque `longjmp()` peut modifier la pile de façon pratiquement arbitraire, il ne peut pas être représentée dans un modèle qui simule la pile d'exécution du programme par un automate à pile. Heureusement, il est possible de détecter les appels à ces fonctions lors de la surveillance d'un programme. Puisque l'algorithme utilisé pour l'analyse des séquences d'appels au noyau permet l'accès à l'ensemble des arbres syntaxiques possibles, il est possible pour le programme de surveillance de modifier en conséquence ces arbres syntaxiques avant de poursuivre son exécution.

5.7.4.6 Autres complications

Les complications suivantes ont été rencontrées lors de l'implémentation du système. D'abord, certaines bibliothèques font un usage important de `longjmp()` et de pointeurs de fonction. L'utilisation de ces deux constructions rend le modèle imprécis parce que leur traitement n'est pas optimal. Dans ces cas, il peut être mieux de construire manuellement un modèle des comportements possibles. Ceci peut faire en sorte que des erreurs se glisse dans le modèle et des fausses alarmes pourraient alors survenir.

Certaines constructions du langage C permettent plusieurs ordres d'évaluation légaux pour certaines expressions. Le modèle doit tous les considérer pour éviter d'être spécifique à un compilateur précis.

L'utilisation de plusieurs fils d'exécution peut aussi poser un problème si leur gestion n'est pas effectuée par le noyau du système d'exploitation.

5.7.4.7 Optimisation

Plusieurs techniques d'optimisation sont suggérées et certaines ont été implémentées par David Wagner pour obtenir une surveillance plus efficace. Parmi elle il y a la réduction du graphe de flot de contrôle. Il est possible d'éliminer les noeuds qui ne font pas d'appel au noyau ni d'appel de fonction. On peut aussi ignorer certains appels au noyau qui ne sont pas intéressant. Par exemple `brk()` est un appel qui permet d'allouer de la mémoire. Son utilisation peut survenir à peu près n'importe où dans un programme. On peut donc l'ignorer pour rendre le modèle plus concis et possiblement améliorer l'efficacité de la surveillance.

Une autre optimisation possible consiste à surveiller les paramètres des appels au noyau en plus des appels eux même. Lorsqu'un paramètre est constant, il est facile de vérifier que c'est le bon paramètre lors de l'exécution. Ceci permet de limiter la taille de l'ensemble des états possibles des différents modèles.

5.8 Analyse statique

Les méthodes statiques offrent beaucoup d'avantages sur les méthodes dynamiques. Elles permettent d'obtenir certains résultats qui sont valables pour toutes les exécutions possibles du pro-

gramme. Cependant, vérifier si un programme permet ou non des débordements de tampons est un problème difficile, non décidable pour le cas général. Il y aura donc toujours des faux positifs ou des faux négatifs pour n'importe quel programme qui tente de dire si un programme arbitraire peut effectuer un débordement de tampon.

Il existe plusieurs approches utilisant l'analyse statique. Certaines ne détectent que les débordements effectués par certaines fonctions, d'autres détectent les débordements n'importe où dans le code. Certaines effectuent l'analyse fonction par fonction, d'autres analysent un programme en entier. Elles se distinguent aussi par le fait qu'elles admettent des faux positifs, des faux négatifs ou les deux ainsi que par la fréquence des faux diagnostiques.

5.8.1 Analyse lexicale

L'analyse lexicale n'est pas une analyse très poussée. Elle permet seulement de reconnaître les unités lexicales du langage avant qu'elles ne forment un arbre syntaxique. Les analyseurs qui utilisent une telle approche sont à la recherche d'une séquence d'unités lexicales qui sont susceptibles de poser des problèmes. Par exemple, un tel analyseur peut rechercher les appels à la fonctions `printf()` qui passent un premier paramètre qui est une variable dans le but de détecter les vulnérabilités de chaîne de format. Pour ce faire, il peut rechercher les deux unités lexicales consécutives «`printf`» et «`(`» et vérifier que l'unité lexicale suivante n'est pas une chaîne de caractères. Il n'essaie pas de déterminer si la variable utilisée vient d'une source fiable ou pas, il indique seulement au programmeur qu'il y a une possibilité de vulnérabilité et il l'incite à inspecter le code pour s'assurer que tout est correct.

Les logiciels qui utilisent cette technique ont l'avantage d'être rapides puisque l'analyse ne demande pas beaucoup de calculs. Ces logiciels donnent habituellement beaucoup de faux positifs et ils ne donnent jamais d'assurance qu'il n'y a aucun débordement possible. Par exemple, ils considèrent que certaines fonctions sont sécuritaires et ne donne pas d'avertissement même si elles sont mal utilisées. Ils ne permettent pas non plus de détecter un débordement provoquer par une mauvaise condition dans une boucle.

L'outil le plus simple permettant d'effectuer une analyse statique est sans doute grep. Cet outil permet de trouver les lignes dans un fichier texte qui contiennent une certaine expression régulière. Par exemple, la commande `grep -n strcpy test.c` permet de trouver toutes les lignes d'un fichier source qui appellent la fonction `strcpy()`, mais aussi toutes celles qui ont une chaîne de caractères ou un commentaire dans lequel se trouve «strcpy». La précision n'est donc pas très bonne, il y a énormément de faux positifs et c'est pourquoi il existe des outils qui tentent de filtrer un peu mieux le code qui est le plus susceptible de poser des problèmes.

Flawfinder, RATS, et ITS4 sont des logiciels qui effectuent une analyse lexicale un peu plus poussée pour augmenter la précision des résultats. Les deux premiers sont des logiciels libres[6] alors que ITS4 ne l'est pas. Son code source est tout de même disponible gratuitement pour usage non commercial.

Flawfinder [Whe02a, Whe02b] permet de détecter des problèmes potentiels de débordement de tampons, des vulnérabilités de chaîne de format et des situations de concurrence[7] dans les

[6]Un logiciel libre peut être librement utilisé, étudié, amélioré et redistribué. `http://www.gnu.org/philosophy/free-sw.fr.html`

[7]Nous utilisons situation de concurrence comme traduction de l'expression anglaiser *race condition*.

programmes C et C++.

RATS [Sec02] permet aussi de détecter des problèmes potentiels de débordement de tampons, des vulnérabilités de chaîne de format et des situations de concurrence dans des programmes C, C++, Perl, PHP et Python.

ITS4 [VBKM01] permet de détecter des problèmes potentiels de débordement de tampons, des situations de concurrence et des mauvaises utilisations de fonction de génération de nombres pseudo-aléatoires dans les programmes C et C++.

5.8.2 Vérification des pré-conditions et des post-conditions

Dans [LE01, EL02], David Larochelle et David Evans expliquent une méthode permettant entre autres choses de détecter les débordements de tampons dans les programmes C au moyen d'une analyse statique légère. Leur méthode évite d'effectuer une analyse interprocédurale en demandant que les pré-conditions et les post-conditions accompagnent le prototype des fonctions. Ces annotations constituent donc une sorte de contrat pour l'utilisation d'une fonction. On peut donc vérifier si le contrat est respecté par chacune des «parties» de façon indépendante.

Pour vérifier si une fonction peut provoquer un débordement de tampon, on présume que les variables qu'elle reçoit en entrée respectent les pré-conditions décrites par les annotations. On peut ensuite vérifier si, sous ces conditions, le code de la fonction manipule les variables d'une manière qui assure que les post-conditions soient respectées au moment où elle se termine. Si ce n'est pas le cas, on peut l'indiquer au programmeur.

Aux endroits où se trouve un appel à une fonction, on vérifie que les paramètres passés à la fonction respectent bien pré-conditions et on suppose pour la suite du traitement que les post-conditions sont respectées. Si une pré-condition n'est pas satisfaite à un point d'appel, on peut l'indiquer au programmeur. Puisque les prototypes sont visibles autant lors de la compilation de des fonctions annotées que lors de la compilation de toutes autres fonctions qui les appellent, on peut être certain que l'analyse effectuée est valable, à moins que les annotations soient modifiées entre la compilation d'une fonction et de son appelant.

L'outil qui permet d'effectuer cette analyse s'appelle Splint et il est un logiciel libre. Son travail ne se limite pas aux débordements de tampons. Il reconnaît aussi des annotations qui permettent de vérifier l'allocation et la libération de la mémoire, les vulnérabilités de chaînes de format et l'utilisation de pointeurs nuls.

Les éléments qui composent principalement les annotations permettant la vérification des débordements de tampons sont les opérateurs `maxSet` et `maxRead`. `maxSet(t)` représente l'indice maximal du tableau t auquel on peut donner une valeur, tandis que `maxRead(t)` représente l'indice maximal du tableau t qui peut être lu. Utilisés dans une clause `requires`, ils permettent d'exprimer les pré-conditions, alors que dans une clause `ensures`, ils expriment les post-conditions. On peut aussi utiliser les opérateurs `minSet` et `minRead` qui représentent l'indice minimal utilisable pour un tableau, soit en écriture ou en lecture. L'exemple 5.3 montre des annotations qui sont tirées de celles fournies avec Splint.

Pour obtenir des résultats utiles de la part de Splint, on n'a pas besoin d'annoter toutes les fonctions. D'abord, il vient avec des déclarations annotées pour les fonctions normalisées de la bi-

```
void strcpy (char *s1, char *s2)
     /*@requires maxSet(s1) >= maxRead(s2) @*/
     /*@ensures maxRead(s1) == maxRead (s2) @*/;

char * fgets (char *s, int n, FILE *stream)
     /*@requires maxSet(s) >= (n -1); @*/
     /*@ensures maxRead(s) <= (n -1) /\ maxRead(s) >= 0; @*/;
```

Exemple 5.3: Extraits d'annotations utilisées par Splint. Seules certaines annotations concernant les débordements de tampon ont été conservées pour montrer le principe général. L'opérateur /\ représente la conjonction.

bliothèque C. Il est donc possible de trouver plusieurs débordements sans ajouter aucune annotation. Ensuite, en l'absence d'annotations, Splint choisie des pré-conditions et des post-conditions par défaut. Ces valeurs par défaut sont définies de façon à correspondre à celles du plus grand nombre possible de fonctions. On peut donc annoter seulement les fonctions qui ne correspondent pas aux conditions par défaut. De plus, puisque Splint effectue une analyse statique légère et qu'il s'exécute rapidement, on peut l'exécuter une fois, modifier quelques annotations puis l'exécuter à nouveau et recommencer autant de fois que nécessaire, de la même façon qu'on corrige les erreurs de syntaxe dans un programme à l'aide d'un compilateur.

Comme nous l'avons indiqué, Splint ne permet pas uniquement la détection des débordements de tampons. Il permet aussi la définition de certains attributs par l'utilisateur. Ces attributs permettent ensuite la vérification de certaines propriétés d'un programme en ajoutant certaines annotations. L'exemple 5.4 montre la définition d'un attribut de «propreté» (`taintedness`) qui permet d'identifier des vulnérabilités de chaîne de format.

```
attribute taintedness
   context reference char *
   oneof untainted, tainted
   annotations
      tainted reference ==> tainted
      untainted reference ==> untainted
      anytainted parameter ==> tainted
   transfers
      tainted as untainted ==> error "Possibly tainted storage used as untainted."
   merge
      tainted + untainted ==> tainted
   defaults
      reference ==> tainted
      literal ==> untainted
      null ==> untainted
end
```

Exemple 5.4: Cet attribut indique si une valeur provient d'une source propre ou pas et il permet la détection de vulnérabilités de chaînes de format.

Cet attribut indique que tout doit être considéré «sale» par défaut, sauf les constantes de chaîne

de caractères. De cette façon, une simple annotation dans la déclaration de la fonction `printf()` (exemple 5.5) permet de détecter des vulnérabilités de chaîne de format dans l'utilisation de cette fonction.

```
extern int printf (/*@untainted@*/ char *format, ...) ;
```

Exemple 5.5: Annotation dans `printf()` pour éviter les vulnérabilités de chaîne de format. On indique simplement que `printf()` doit prendre une chaîne de format «propre».

L'analyse effectuée par Splint n'est pas saine ni complète. En effet, elle peut provoquer des faux positifs et des faux négatifs. Parfois l'analyse n'est pas assez poussée pour permettre de déterminer qu'un bout de code respecte certaines conditions. L'analyse tient compte du flot de contrôle, mais d'une manière minimale. Elle ne permet pas non plus de toujours vérifier que le résultat d'une certaine expression est dans un certain intervalle.

D'autres fois, certaines simplifications effectuées font en sorte que certaines formes de non respect des conditions peuvent échapper à l'analyse. L'analyse est effectuée instruction par instruction et elle ne considère pas que la valeur d'une variable puisse changer pendant une instruction. Pour l'analyse d'une boucle, des heuristiques sont utilisées pour déterminer le nombre de fois qu'elle peut s'exécuter, mais elles ne garantissent pas que le résultat soit bon. L'analyse suppose qu'une boucle qui provoque un débordement de tampon montre ce comportement lors de la première ou de la dernière itération. C'est le cas la plupart du temps, mais il est possible de construire une boucle qui effectue un débordement sans que ce soit dans la première ni la dernière itération.

Splint ne permet donc pas de garantir qu'il n'y a pas de débordements, mais pour la plupart des programmes réels, il devrait permettre de pratiquement tous les détecter. Par contre, il donne beaucoup de faux positifs qui doivent être vérifiés manuellement par le programmeur. Il est cependant utile. Par exemple, pour le programme WU-FTPD, l'ajout de 22 annotations a suffit pour permettre à Splint de déterminer automatiquement que 92% des 225 appels à des fonctions considérées généralement comme dangereuses étaient corrects.

5.8.3 Type abstrait et résolution de contraintes

Dans [WFBA00, Wag00], David Wagner considère les chaînes de caractères comme un type abstrait manipulé par les fonctions de la bibliothèque C. Les chaînes de caractères sont modélisées par une paire d'entiers qui indiquent l'espace allouée pour la chaîne et sa longueur. À chaque manipulation d'une chaîne de caractères sont associées une ou plusieurs contraintes sur ces deux entiers.

Le temps mis pour la résolution d'un système de contraintes avec l'algorithme décrit est proportionnel au nombre de contraintes pour les cas observés. Le résultat est, pour chaque chaîne, un intervalle de valeurs possibles pour la mémoire allouée et un autre intervalle pour la longueur. Pour être certain qu'il n'y a pas de débordement de tampons, il faut que la borne supérieur de l'intervalle représentant la longueur de la chaîne soit plus petite ou égale à la borne inférieure de l'intervalle qui représente la quantité de mémoire allouée.

L'outil qui implémente cette technique s'appelle BOON. Il est écrit en ML et une partie de son code est libre. Il comporte cependant des limites importantes. D'abord, il ne peut détecter les

débordements que sur les chaînes de caractères et pas sur les tableaux arbitraires. De plus, pour les chaînes de caractères, seules les manipulations par les fonctions de la bibliothèque C sont considérées. Il y a aussi la fonction `strncpy()` qui n'est pas modélisée correctement, il ne traite pas le cas où la chaîne résultante n'est pas terminée par un caractère nul.

La gestion des pointeurs pose aussi des problèmes car elle est très simplifiée dans BOON. Par exemple, il ignore en grande partie le fait que deux pointeurs peuvent référencer à la même chaîne de caractères. Il ignore aussi les pointeurs à indirection double, les tableaux de pointeurs, les pointeurs de fonctions et les `union`. Toutes ces lacunes font en sorte que certains débordements ne sont pas détectés. L'exemple 5.6 montre un cas de débordement qui ne serait pas détecté par BOON.

```
char s[20], *p, t[10];
strcpy(s, "Hello");
p = s + 5;
strcpy(p, " world!");
strcpy(t, s);
```

Exemple 5.6: Le débordement du tampon t n'est pas détecté par BOON parce que la chaîne s est modifiée par un autre pointeur.

L'analyse effectuée pour générer les contraintes a l'avantage d'être rapide, mais elle manque de précisions. Ceci a pour effet que le nombre de faux positifs est plutôt élevé avec BOON. D'abord l'analyse ne tient pas compte du flot de contrôle, c'est-à-dire que l'ordre des instructions est ignoré, de même que les structures de contrôle. Ensuite, l'outil considère qu'il n'y a qu'une instance de chacune des variables composant une structure, plutôt qu'une par instance de la structure. Ceci peut élargir grandement l'intervalle des valeurs possibles calculé par l'outil pour une variable par rapport à l'intervalle qu'elle peut vraiment prendre. Ce principe a par contre l'avantage de considérer que deux pointeurs à une structure peuvent modifier une même variable.

BOON a permis de découvrir des débordements auparavant inconnus dans quelques logiciels utilisés sous Linux. Bien que le nombre de faux positifs soit élevé, il permet de diminuer leur d'un facteur d'environ 15 comparativement à la simple utilisation de grep.

Chapitre 6

Conclusion

Dans ce rapport, nous avons étudié ce que sont les débordements de tampons et nous avons vu comment ils peuvent être exploités. Nous avons aussi vu comment une simple chaîne de format pouvait devenir une source de vulnérabilité très importante. Nous avons examiné des méthodes pour détecter ces vulnérabilités, s'assurer qu'elles n'existent pas et éviter leur exploitation. L'idéal serait d'avoir une approche statique qui donne l'assurance qu'un programme ne présente pas de telles vulnérabilités sans même l'exécuter. Le recours à des méthodes dynamiques pour les empêcher ou limiter leurs conséquences deviendrait alors superflu et le programme pourrait s'exécuter à sa pleine vitesse sans risque de compromettre la sécurité d'un système.

Nous sommes par contre très loin de cet idéal. Les outils disponibles présentement et qui utilisent une approche statique ont des limites importantes qui empêchent d'avoir une pleine confiance en la validité des résultats. Les outils dynamiques quant à eux donnent un résultat qui n'est valide que pour une exécution du programme, et la plupart d'entre eux ne garantissent pas qu'ils empêchent tous les débordements de tampons. Ceux qui offrent cette garantie ont un coût très élevé en temps d'exécution.

De plus, il reste toujours un problème : quoi faire lors de la détection d'une vulnérabilité ? Le mieux qu'on puisse faire est d'arrêter le programme puisque le laisser s'exécuter dans un état incohérent risquerait de mener à des conséquences plus graves encore. Par contre, il est loin d'être souhaitable d'arrêter un programme qui pourrait être critique dans un système.

Il semble que la protection contre les vulnérabilités de chaîne de format soit plus facile que la protection contre les débordements de tampons, du moins si l'on accepte une protection lors de l'exécution et qu'il est possible de recompiler le programme qui doit être protégé. Bien que la protection offerte par FormatGuard ne soit théoriquement pas à toute épreuve, en pratique elle permet d'attraper pratiquement toutes les vulnérabilités et elle n'impose pas une dégradation significative de la performance.

Au cours de ce rapport, nous avons examiné surtout des outils qui s'appliquent aux langages C et C++. Si plusieurs d'entre eux s'appliquent indifféremment à ces deux langages et que d'autres sont indépendants du langage utilisé, il reste que plusieurs approches permettent de traiter le C et pas le C++. Ceci s'explique surtout par le fait que le C++ est un langage beaucoup plus complexe que le C.

Annexe A

Exploitation d'une vulnérabilité de chaîne de format

Cette annexe décrit de façon détaillée une vulnérabilité de chaîne de format présente dans les versions antérieures à 2.6.1 du serveur FTP WU-FTPD et un programme qui permet de l'exploiter. Ce dernier est connu sous le nom de 7350wu [sz00]. Il utilise la force brute basée sur la réponse (mentionné à la section 4.3) pour trouver toutes les adresses exactes nécessaires à l'exploitation de WU-FTPD.

Cette annexe est basé sur [scu01, sz00, WU-b, WU-a].

A.1 Description de la vulnérabilité

Le serveur FTP WU-FTPD est écrit en C et il définit certaines fonctions qui prennent en paramètre une chaîne de format. Avant la version 2.6.1, WU-FTPD avait un bogue qui faisait en sorte qu'il pouvait passer à une de ses fonctions une chaîne de caractères qui provenait directement de l'utilisateur. L'exemple A.1 montre où se trouve la vulnérabilité. La figure A.1 montre l'organisation de la pile au moment où la vulnérabilité de format se produit.

Le bogue se trouve dans la fonction `site_exec()`. Celle-ci devrait passer une chaîne de format à `lreply()` plutôt qu'une chaîne arbitraire. On peut corriger le problème en passant la chaîne de format `"%s"` avant l'autre chaîne de caractères. L'exemple A.2 montre l'extrait corrigé.

La fonction `site_exec()` implémente une extension au protocole FTP. Cette extension peut être utilisée au moyen de la commande `SITE EXEC`. `SITE` est une commande FTP qui est réservée pour que les serveurs FTP puissent offrir des fonctionnalités supplémentaires aux fonctionnalités de base définies par le protocole. Avec WU-FTPD, `SITE EXEC` permet à un utilisateur distant d'exécuter un programme sur le serveur et d'obtenir pour résultat la sortie de ce programme. Bien entendu, seuls les programmes choisis par l'administrateur du serveur FTP peuvent être exécutées par un utilisateur distant. Il doit les places dans un répertoire spécial qui est normalement le sous-répertoire bin/ftp-exec du répertoire de base du compte ftp. Le plus souvent, ce répertoire n'existe pas ou il

Extraits de ftpcmd.c.

```
void site_exec(char *cmd)
{
    char buf[MAXPATHLEN];
...
    sprintf(buf, "%s/%s", _PATH_EXECPATH, cmd);
...
        lreply(200, cmd);
        while (fgets(buf, sizeof buf, cmdf)) {
...
            lreply(200, buf);
...
        }
...
}
```

Extraits de ftpd.c.

```
void lreply(int n, char *fmt,...)
{
    VA_LOCAL_DECL
...
    VA_START(fmt);

    /* send the reply */
    vreply(USE_REPLY_LONG, n, fmt, ap);

    VA_END;
}


void vreply(long flags, int n, char *fmt, va_list ap)
{
    char buf[BUFSIZ];
...
    if (flags & USE_REPLY_NOTFMT)
        snprintf(buf + (n ? 4 : 0), n ? sizeof(buf) - 4 : sizeof(buf), "%s", fmt);
    else
        vsnprintf(buf + (n ? 4 : 0), n ? sizeof(buf) - 4 : sizeof(buf), fmt, ap);
...
    printf("%s\r\n", buf);      /* and send it to the client */
}
```

Exemple A.1: Extraits du code source de WU-FTPD 2.6.0. Il y a une vulnérabilité de chaîne de format dans la fonction `site_exec` car *cmd* provient directement de l'utilisateur.

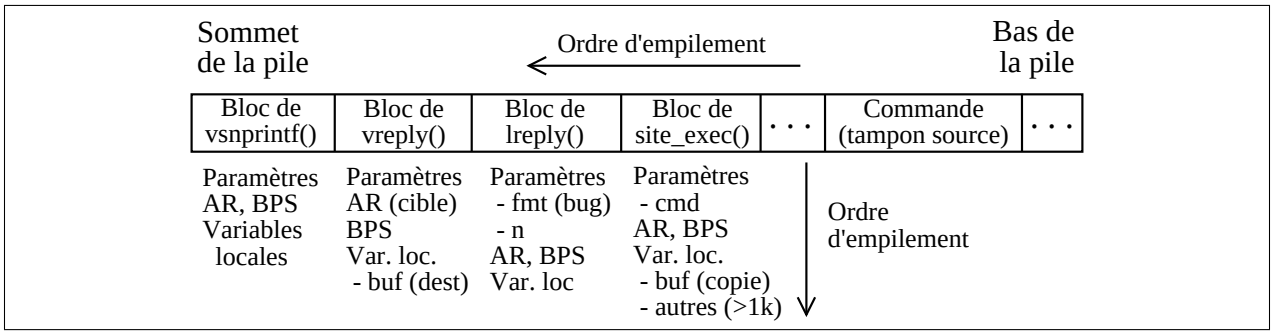

Figure A.1: Pile de WU-FTPD au moment de la vulnérabilité. Cette illustration complète les extraits de code de l'exemple A.1. Les mentions entre parenthèses permettent de voir les éléments critiques de l'exploitation. (bug) indique qu'une chaîne arbitraire est passée à la place d'une chaîne de format.

```
void site_exec(char *cmd)
{
...
        lreply(200, "%s", cmd);
        while (fgets(buf, sizeof buf, cmdf)) {
...
            lreply(200, "%s", buf);
...
        }
...
}
```

Exemple A.2: Code source WU-FTPD corrigé

est vide et aucun programme n'est accessible par l'utilisateur distant au moyen de la commande `SITE EXEC`.

Cependant, la vulnérabilité de chaîne de format montrée à l'exemple A.1 peut être exploitée même lorsque aucun programme n'est exécutable. En effet, si le programme dont l'utilisateur demande l'exécution n'existe pas ou n'est pas exécutable, WU-FTPD tente quand même de l'exécuter et tout se passe normalement, sauf que le programme ne génère évidement pas de sortie. L'exemple A.3 montre une session FTP où `SITE EXEC` est utilisé. Il démontre aussi que la vulnérabilité de chaîne de format est bel et bien réelle.

```
$ ftp zero
Connected to zero.
220 zero FTP server (Version wu-2.6.0(1) Mon Dec 2 21:49:17 EST 2002) ready.
Name (localhost:tootix): ftp
331 Guest login ok, send your complete e-mail address as password.
Password:
230 Guest login ok, access restrictions apply.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> quote SITE EXEC essai
200-essai
200  (end of 'essai')
ftp> quote SITE EXEC %08X hé!
200-080678c2 hé!
200  (end of '%08x hé!')
ftp> quit
221-You have transferred 0 bytes in 0 files.
221-Total traffic for this session was 414 bytes in 0 transfers.
221-Thank you for using the FTP service on zero.
221 Goodbye.
```

Exemple A.3: Démonstration de l'utilisation de `SITE EXEC`. Avec ce client FTP, il faut utiliser la commande `quote` pour passer la commande `SITE EXEC` au serveur. Ici le programme `essai` n'existe pas. On voit aussi la vulnérabilité de chaîne de format en action.

A.2 Description d'une attaque : 7350wu

De nombreuses attaques permettant d'exploiter la vulnérabilité de chaîne de format de WU-FTPD ont été publiée depuis juin 2000, [tf800, Bou00b] ne sont que deux exemples. Nous avons choisi de présenter 7350wu parce qu'il utilise une technique que nous appelons force brute basée sur la réponse. Contrairement aux autres attaques publiées où il faut «deviner» à l'avance l'adresse exacte de certaines variables en mémoire, cette technique permet de rechercher efficacement ces adresses. Ceci est possible parce que la chaîne de format qu'on envoie au serveur est non seulement traitée par le serveur, mais le résultat formaté est aussi retourné au client.

On peut compiler 7350wu [sz00] en utilisant simplement la commande `make` après avoir dé-

compressé l'archive. Les sections suivantes décrivent le fonctionnement des différentes phases de l'attaque telle qu'implémentée par 7350wu.

A.2.1 Ouverture de session

La première étape effectuée par 7350wu est l'ouverture d'une session avec le serveur FTP. C'est la fonction `ftp_login()` qui effectue l'ouverture de la session. On peut spécifier à 7350wu le nom du serveur visé au moyen de l'option «`-h`». Par défaut il se connecte de façon anonyme avec le nom d'utilisateur «`ftp`» et le mot de passe «`mozilla@`». Il est possible d'utiliser d'autres valeurs pour le nom d'utilisateur et le mot de passe en utilisant les options «`-u`» et «`-p`» respectivement.

A.2.2 Vérification de la vulnérabilité

La deuxième étape est de vérifier si le serveur est vulnérable. C'est la fonction `ftp_vuln()` qui effectue le test de vulnérabilité. Elle envoie la commande suivante :

```
SITE EXEC %020d|%.f%.f|
```

S'il y a une vulnérabilité de chaîne de format, «`%020d`» devrait être remplacé par un nombre de 20 chiffres avec des 0 pour le remplissage. La fonction vérifie que la réponse débute par `200-0000000`. Les 3 premiers caractères, `200` dans ce cas-ci, représentent le code de retour, 200 étant «Ok». Le caractère suivant, «`-`» indique que la réponse se poursuit sur la ligne suivante. Il est donc présent sur toutes les lignes de réponse à une commande sauf la dernière. Puisque la chaîne formatée est transmise sur la première ligne de la réponse, 7350wu ne tient compte que de la première ligne des réponses qu'il reçoit et ignore les autres.

Les caractères qui suivent le «`-`» sont les 0 de remplissage attendus suite à l'interprétation de la chaîne de format. La fonction vérifie également que la réponse ne contient pas la chaîne `|????????????????|`. Ceci signifierait que la fonction `vsnprintf()` est très minimale et qu'elle ne supporte pas les spécificateurs de conversion «`%.f`» utilisés par cette attaque.

A.2.3 Recherche de la distance du tampon contrôlé

L'étape suivante permet de trouver la distance sur la pile entre le premier paramètre variable présumé et la position de la chaîne de caractère interprétée comme une chaîne de format sur la pile. La connaissance de cette distance permet de savoir combien de paramètres il faut «brûler» avant d'atteindre un espace contrôlé sur la pile.

Pour ce faire, la fonction `ftp_finddist()` envoie une commande de la forme :

SITE EXEC 7 `mmmmnnnn`*popstack*`|%08x|%08x|`

Ici, «`mmmmnnnn`» est un marqueur, *popstack* est une séquence de «`%.f`» qui permettent chacun de sauter 8 octets de la pile. Les «`%08x`» permettent d'afficher la valeur du mot visé sur la pile. Le but est qu'un de ces deux paramètres de conversion soit remplacé par la séquence «`6d6d6d6d`» qui est la représentation hexadécimale de «`mmmm`». Les barres verticales permettent de retrouver ces valeurs dans la sortie. Si la valeur cherchée suit la première barre verticale, ça signifie que la séquence

popstack utilisée pour atteindre la valeur peut être utilisée telle quelle dans la suite de l'attaque. Si elle est trouvée à la suite de la deuxième barre verticale, il faut ajouter un spécificateur de conversion «`%d`» à la séquence de manière à passer 4 octets supplémentaires.

Si la valeur cherchée n'est pas trouvée, un «`%.f`» supplémentaire est ajouté à *popstack* de manière à explorer ce qui se trouve 8 octets plus loin et la recherche continue à cet endroit.

La fonction `ftp_finddist()` effectue aussi un certain traitement pour trouver la position du marqueur s'il n'est pas aligné sur un multiple de 4 octets. C'est ce qui explique la présence des `nnnn` dans le marqueur. Bien que la position exacte du marqueur soit trouvée correctement, le reste de l'attaque ne traite pas correctement ce cas. Il faut dire qu'à moins que le code source de WU-FTPD ne soit modifié afin d'utiliser un répertoire différent pour les programmes exécutables au moyen de la commande `SITE EXEC`, le préfixe choisi dans l'attaque garanti que le marqueur sera toujours aligné.

Ce n'est pas dans le tampon destination de `vreply()` que le marqueur est retrouvé parce qu'il se trouve à une adresse plus basse que celle des paramètres variables de `lreply()` sur la pile. Ce n'est pas non plus dans le tampon source (celui de la chaîne de commande) qu'il est retrouvé, mais plutôt dans un tampon de `site_exec()` où une partie de la chaîne de commande est copiée. Ce tampon a l'avantage d'être beaucoup plus près des paramètres de `lreply()` et l'atteindre demande donc une séquence *popstack* plus courte.

La recherche ne débute pas immédiatement aux paramètres variables de `lreply()`, mais plutôt 1024 octets plus loin. C'est parce que la fonction `site_exec()` définit entre autres un tableau de 1024 octets qui se trouve sur la pile avant le tampon de copie. Éviter d'effectuer la recherche dans cette partie de la mémoire permet de trouver plus rapidement le marqueur recherché.

A.2.4 Recherche de l'adresse du tampon source

Après avoir trouvé une séquence *popstack* qui «mène» à une zone contrôlée, l'attaque se poursuit en cherchant l'adresse du tampon source, celui qui contient la commande envoyée par l'utilisateur. Pour ce faire, la fonction `ftp_findaddr()` utilise une commande de la forme :

`SITE EXEC 7` *addr popstack bips* `%%|x|%.`*n*`s`

Ici *addr* est l'adresse en format binaire que l'on soupçonne être dans le tampon de la chaîne de commande et *popstack* est la séquence qui mène les paramètres variables dans la zone contrôlée, exactement à l'endroit où se trouve *addr*. Puisque le spécificateur suivant est (une variante de) «`%s`», le contenu de la mémoire qui se trouve à cette adresse est lu et retourné dans la réponse. Quant à *bips*, il est composé d'un certain nombre de caractères «`_`» (bip) pour obtenir une commande de 512 octets. Finalement, *n* correspond au nombre de «bips» et nous allons voir pourquoi plus tard.

L'adresse *addr* ne peut pas contenir de caractère nul «`\0`» ni de caractère de changement de ligne «`\n`» parce que ces caractères termineraient la commande. Il n'est pas non plus possible d'utiliser un «`%`» parce qu'il serait interprété dans la chaîne de format. C'est la fonction `esc_ok()` qui effectue ces vérifications. Si l'adresse ciblée contient un de ces caractères, elle est simplement passée et la recherche continue à l'adresse suivante.

Si *addr* contient un ou des caractères «`\xff`», ils doivent être doublés car ils servent de séquence

d'échappement pour le protocole FTP. La fonction `xpad_cat()` est utilisée pour ajouter une adresse dans ce format à une chaîne de caractères.

La séquence «`%%|x|`» qui devient «`%|x|`» dans la réponse retournée par le serveur, permet d'identifier facilement où débute le contenu de la mémoire à l'adresse *addr*. Pour plus de précision, c'est la séquence «`_%|x|`» qui est recherchée. Pour que l'adresse ciblée soit considérée bonne, elle doit faire partie des *bips*. Dans ce cas, on peut compter le nombre de bips consécutifs retournés par le serveur à la suite de «`_%|x|`». La formule permettant de calculer l'adresse de début du tampon de la chaîne source est donc

$$addr + nb\_bips - lng\_bips$$

où *lng_ bips* est la longueur de la portion de la commande (envoyée au serveur) se terminant à la fin des bips et *nb_ bips* est le nombre de bips retournés dans la réponse.

C'est le spécificateur de conversion «`%.`*n*`s`» qui permet d'obtenir le contenu de la mémoire à l'adresse visée. La valeur de *n* indique le nombre maximal de caractères demandés et elle correspond au nombre de bips parce que par la suite on doit compter leur nombre et on ne peut pas en trouver plus que le nombre placé dans la commande. On évite ainsi d'obtenir une réponse inutilement grande de la part du serveur dans le cas où on toucherait à une longue chaîne de caractères.

La recherche de l'adresse du tampon source se fait à partir des adresses élevées vers les adresses plus basses. De cette façon, la recherche passe par le tampon source (plus haut en mémoire) avant d'atteindre le tampon destination. On peut donc conclure que la première «bonne» adresse trouvée est dans le tampon source. À chaque fois que *addr* manque la cible, on la décrémente du nombre de bips placés dans la commande et on recommence. Donc plus il y a de bips dans la commande, plus la recherche peut se faire rapidement.

A.2.5 Recherche de l'adresse du tampon destination

La recherche de l'adresse du tampon destination est très similaire à celle du tampon source. Elle est effectuée par la fonction `ftp_finddaddr()` qui effectue une vérification supplémentaire permettant d'obtenir l'assurance que l'adresse pointe bel et bien dans le tampon destination plutôt que dans le tampon source ou une copie de celui-ci. La commande utilisée pour la recherche du tampon de destination a exactement le même format que celui utilisé pour la recherche du tampon source.

Pour être certain que c'est le tampon destination qui a été touché, on vérifie non seulement que ce qui suit la séquence «`_%|x|` est un ou plusieurs «_» (bips) mais également que ces bips se terminent par la séquence «`_%|x|`». Si c'est le cas on est certain d'avoir atteint le tampon destination puisque les «%%» ont été transformés en «%» par `vsnprintf()`. Sinon on avait trouvé le tampon source ou une copie de ce dernier, on trouverait plutôt «_%%|x|» dans le tampon.

Puisqu'il faut vérifier plus de caractères que dans le cas du tampon source et qu'on ne veut pas allonger la taille de la réponse, on utilise un décrément plus petit que celui utilisé dans la recherche du tampon source. La valeur utilisée est 16 de moins que le nombre de bips.

La formule qui permet de calculer l'adresse du tampon destination est :

$$addr + nb\_bips - lng\_bips$$

où *addr* est l'adresse qui a permis d'atteindre les bips du tampon destination, *lng_bips* est la longueur de la portion de la réponse du serveur se terminant à la fin des bips et *nb_bips* est le nombre de bips retournés dans la réponse.

A.2.6 Calcul de l'adresse de retour

C'est en écrasant une adresse de retour que cette attaque permet d'exécuter du code arbitraire. Le calcul de l'adresse de retour est plutôt simple. Le code qui doit être exécuté se trouve dans une chaîne de caractères appelée *shellcode*. Celui-ci sera placé à la fin de la chaîne de commande qui a une longueur de 511 caractères si on ne compte pas le caractère de fin de chaîne. L'adresse de retour à utiliser est donc

$$addr_{ret} = addr_{source} + 511 - \texttt{strlen}(shellcode)$$

où $addr_{source}$ est l'adresse trouvée par la démarche décrite à la section A.2.4. Il faut noter que c'est cette adresse qui sera écrite en mémoire plus tard au moyen de «`%n`» et que les caractères spéciaux ne posent aucun problème.

Par mesure de précaution, quelques `NOP` sont ajoutés avant *shellcode* dans la chaîne de commande et quelques octets sont soustraits pour obtenir la véritable adresse de retour. Ceci permet de tenir compte des caractères «\xff» qui sont doublés dans la «chaîne de commande» transmise au serveur mais qui ne le sont plus dans le tampon source du côté du serveur.

A.2.7 Recherche de la position de l'adresse de retour

L'étape suivante permet de localiser la position en mémoire de l'adresse de retour à écraser. La fonction `main()` tente de deviner cette position, que nous appelons *retloc*, en ajoutant 8196 à l'adresse du tampon de destination. On calcule 8192 octets pour le tampon destination et 4 octets pour le pointeur de bloc de pile sauvegardé. De cette façon, c'est l'adresse de retour de la fonction `vreply()` qui est visée. Si la tentative s'avère inexacte, on regarde «autour» de cette adresse, un peu plus haut et un peu plus bas.

Pour décider si l'adresse calculée est la bonne ou non, on obtient les 4 octets qui s'y trouvent et on vérifie qu'ils correspondent à une adresse dans l'intervalle $[0804000_{16}, 08060000_{16}]$. Cet intervalle est approximativement celui de la zone de mémoire dans laquelle le code de WU-FTPD est chargé.

C'est la fonction `ftp_findrl()` qui permet d'obtenir le contenu d'une adresse. Pour ce faire, elle utilise une commande de la forme :

`SITE EXEC 7` *addr popstack* `|%.4s|`

Ici *addr* est l'adresse dont on veut obtenir le contenu. Dans la réponse du serveur, les 4 caractères qui suivent la première barre verticale rencontrée sont interprétés comme le contenu de l'adresse cible, à moins qu'une barre verticale ne fasse partie de ces caractères. La présence d'une barre verticale peut signifier qu'un des 4 octets se trouvant à l'adresse visée est nul. Dans ce cas l'adresse n'est pas utilisable pour l'attaque. Par contre, ça pourrait aussi vouloir dire qu'un des octets est réellement la barre verticale et cette possibilité n'est pas considérée.

L'écrasement de l'adresse de retour s'effectue à une adresse provenant directement de la chaîne de commande. Il n'est donc pas possible d'écraser à une adresse comportant des caractères tels que «\0», «\n» ou «%».

A.2.8 Le coup de grâce

Une fois toutes les étapes préliminaires effectuées, il ne reste plus qu'à envoyer la commande qui permet d'écraser l'adresse de retour et d'exécuter du code arbitraire. Cette commande a la forme :

`SITE EXEC 7` *retseq popstack writeseq shellcode*

Ici, *writeseq* est formée de quatre parties ayant la forme «`%`w_i`d%n`». Chacune de ces parties permet d'écrire un mot de 32 bits dont l'octet de poids faible correspond à un des octets de $addr_{ret}$. Cette technique est expliquée à la section 4.2.2. Les mots sont écrits à des adresses consécutives se trouvant dans *retseq* de façon à ce que le résultat soit $addr_{ret}$. C'est la valeur des w_i (un entier) qui permet de déterminer combien de caractères sont «sortis» lors de la conversion d'un entier et, du même coup, de contrôler la valeur du mot écrit. Ce mot correspond au nombre de caractères «sortis» au moment où le «`%n`» est rencontré. Chaque w_i est donc choisi de manière à ce que l'octet de poids faible du nombre de caractères coïncide avec un des octets de l'adresse de retour.

Plus précisément, *retseq* est composé de 7 mots de 32 bits. Les premier, troisième, cinquième et septième sont respectivement $retloc$, $retloc + 1$, $retloc + 2$ et $retloc + 3$. Ils sont utilisés par les spécificateurs de conversion «`%n`» de *writeseq*. Les deuxième, quatrième et sixième sont de mots qui prennent la valeur 0x73507350 dans cette attaque, mais qui pourraient être remplacé par à peu près n'importe quelle autre valeur qui peut être placée dans la commande. Ils sont interprétés par les trois derniers spécificateurs de conversion «`%`w_i`d`», leur valeur n'est donc pas importante. Le premier de ces spécificateurs de conversion quant à lui est pris sur la pile juste avant *retseq* pour économiser de l'espace dans la chaîne de commande. La séquence *stackpop* est donc modifiée pour lire 4 octets de moins de la pile que dans les commandes précédentes.

Pour être en mesure de trouver la valeur exacte à utiliser pour les w_i, il faut d'abord savoir combien de caractères sont générés par la partie de la commande qui précède *writeseq*. Pour ce faire la fonction `ftp_getwlen()` envoie une commande qui respecte le même format vu précédemment pour ce qui précède *writeseq* et compte le nombre de caractères obtenus dans la réponse du serveur. Quatre caractères sont soustraits parce qu'il ne faut pas compter «`200-`» qui fait parti de ce qui est ajouté par le protocole FTP.

Finalement, *shellcode* est le code machine qui doit être exécuté sur le serveur. Des `NOP` sont insérés au début de *shellcode* pour que la commande remplisse complètement le tampon prévu pour l'envoie. Le *shellcode* utilisé dans l'attaque est spécial dans le sens que la seule chose qu'il fait est de lire et exécuter la deuxième partie du code servant à l'attaque. Le *shellcode* est donc très petit (seulement 16 octets incluant un «\xff» qui doit être doublé !) ce qui est nécessaire puisque l'espace est très limité dans le tampon. Il permet de lire jusqu'à 255 octets provenant de la connexion réseau avec 7350wu. La deuxième partie, *shellcode2*, permet par la suite de démarrer un interpréteur de commandes sous le compte «root».

À partir de ce point, 7350wu devient interactif. Les commandes que l'utilisateur entre sont passées directement au serveur et tout ce que le serveur (ou l'interpréteur de commandes !) retourne

est affiché à l'utilisateur. C'est exactement comme si l'utilisateur avait démarré une session telnet, mais sans avoir besoin de mot de passe.

A.2.9 Problèmes de 7350wu

7350wu ne fonctionne pas à tout coup, c'est-à-dire qu'il existe des systèmes sur lesquels la vulnérabilité est présente mais où l'attaque ne fonctionne pas. Deux choses importantes sont particulièrement sujettes à faire échouer l'attaque. Premièrement, si le système d'exploitation ne permet pas l'exécution de code sur la pile, l'attaque ne fonctionne pas. En effet, le code à exécuter est placé dans le tampon source qui est sur la pile. Pour contourner ce problème il faudrait ajouter une étape supplémentaire pour la copie du code sur le tas. De cette façon il ne serait plus nécessaire d'utiliser *shellcode*, seul *shellcode2* suffirait.

L'autre problème demande plus d'explications. Il faut savoir que le serveur s'exécute habituellement dans une sorte de prison[1] lorsque l'utilisateur est connecté de façon anonyme. Dans cette situation il n'y a que les fichiers qui se trouvent dans le compte de l'utilisateur FTP qui sont accessibles. En particulier, l'interpréteur de commandes n'est pas disponible, ce qui n'est pas très pratique pour l'attaquant. Le code qu'il exécute doit donc être en mesure de sortir de la prison avant d'exécuter l'interpréteur de commandes. L'exemple A.4 montre la méthode utilisée par 7350wu pour sortir de la prison et exécuter l'interpréteur de commandes.

```
mkdir("bin")
chroot("bin")
Faire 255 fois:
        chdir("..")
chroot(".")
execve("/bin/sh")
```

Exemple A.4: Méthode utilisée par 7350wu pour sortir de prison. Ceci est l'algorithme correspondant à une partie du code machine se trouvant dans la variable *x86_lnx_shell* (*shellcode2*).

Chaque `chdir("..")` permet de remonter d'un niveau dans l'arborescence. Une fois à la racine, les `chdir("..")` supplémentaires ne modifient pas le répertoire courant. Du moins c'était comme ça avant. Avec les dernières versions de Linux 2.4, si un programme tente de remonter à un niveau supérieur à la vrai racine alors qu'il y a une prison `chroot` définie, il ne reste pas à la racine et il n'a plus accès à aucun fichier par un chemin relatif. Si on connaît le nombre de répertoire qu'il faut remonter pour atteindre la racine on peut simplement remplacer 255 par ce nombre. Par exemple, si le compte du serveur FTP se trouve dans le répertoire /home/ftp, on n'a qu'à utiliser 2 à la place de 255 et il est possible de sortir du `chroot`. L'exemple montre comment on peut modifier le code pour sortir d'un `chroot` dans un répertoire de deuxième niveau.

Lorsqu'on n'est pas connecté de façon anonyme, le serveur FTP ne crée pas de prison `chroot`, il n'y a donc rien de spécial à faire pour en sortir. Le code de *x86_lnx_shell* pourrait être simplifié grandement pour ce cas, mais il est possible d'obtenir un comportement correct avec très peu de

[1] En anglais on utilise souvent *chroot jail*. `chroot` est un appel au noyau qui permet de confiner un programme à un certain sous-répertoire. Les utilisateurs non privilégiés ne peuvent pas utiliser `chroot` habituellement.

Pour sortir d'un `chroot` dans un répertoire de deuxième niveau, on peut remplacé la 6e ligne de *x86_ lnx_ shell* :

```
"\x43\x02\x31\xc9\xfe\xc9\x31\xc0\x8d\x5e\x08\xb0"
```

par celle-ci :

```
"\x43\x02\x31\xc9\xb1\x02\x31\xc0\x8d\x5e\x08\xb0"
```

On remarque que seul deux octets ont été modifiés. Dans la séquence originale «`\xfe\xc9`» correspond à l'instruction `dec %cl`. Celle-ci étant exécutée alors que `ECX` vaut 0, la boucle de `chdir` s'exécute 255 fois.
Dans la séquence modifiée, «`\xb1\x02`» correspondent à l'instruction `mov $0x2,%cl`. Puisque `ECX` vaut toujours 0 la boucle de `chdir` s'exécute 2 fois. On pourrait évidemment utiliser une valeur différente de «`\x02`» si le répertoire du `chroot` était à un niveau différent.

Exemple A.5: Sortir d'un `chroot` à un niveau connu.

changements. L'exemple A.6 montre ce qui doit être modifié pour éviter le premier `chroot`. De cette façon tous les `chdir` ont un comportement bien défini et le dernier `chroot` ne fait pas de mal.

Pour éviter le premier `chroot` de *x86_ lnx_ shell* il suffit de remplacer la 5e ligne :

```
"\xb0\x3d\xcd\x80\x31\xc0\x31\xdb\x8d\x5e\x08\x89"
```

par celle-ci :

```
"\xb0\x3d\x90\x90\x31\xc0\x31\xdb\x8d\x5e\x08\x89"
```

Encore une fois, seul deux octets ont été modifiés. Dans la séquence originale «`\xcd\xc80`» correspond à l'instruction `int $0x80` qui déclenche un appel au noyau sous Linux. Elle est remplacée par deux «`\x90`» qui sont des NOP.

Exemple A.6: Éviter le premier `chroot` pour un compte réel.

Si on voulait utiliser du code qui fonctionne autant lorsqu'on utilise un compte réel qu'un compte anonyme, et peu importe le niveau du `chroot` il serait possible d'écrire du code machine équivalent à l'algorithme donné à l'exemple A.7.

Il existe aussi d'autres alternatives pour sortir d'une prison `chroot`. Dans [Ano02] on explique qu'il est possible de prendre le contrôle d'un programme qui se trouve à l'extérieur de la prison au moyen de `ptrace`, un appel au noyau permettant le débogage.

```
Faire pour toujours:
        execve("/bin/sh")
        mkdir("bin")
        chroot("bin")
        chdir("..")
        chroot(".")
```

Exemple A.7: Sortir d'un chroot, prise 2. Cet algorithme est une variante de celui présenté à l'exemple A.4 adaptée pour les dernières versions de Linux 2.4

Annexe B

Logiciels intéressants

Cette annexe présente différents programmes qui peuvent être utile pour détecter ou éviter les débordements de tampons ou les vulnérabilités de chaîne de format ou certaines de leurs conséquences. Nous n'avons pas ou très peu testé la plupart de ces programmes et l'information qui se trouve dans cette section est basée sur la documentation disponible. Ils sont classés par catégorie.

Pour chaque programme, on classe la licence dans une des catégories suivantes.

- Libre : le logiciel est libre (voir la section 5.8.1).
- Gratuit : le logiciel est disponible gratuitement.
- Commercial : le logiciel n'est pas libre, ni gratuit.

B.1 Analyse statique

Cette section décrit des outils d'analyse statique qui peuvent être utiles pour la détection de débordement de tampons ou de vulnérabilités de chaîne de format. Cette liste n'est pas exhaustive et on trouve une liste d'outils d'analyse statique dont certains permettent la détection de débordement de tampons dans [Mat01].

Splint (Libre) Nous avons discuté de ce programme et de son fonctionnement à la section 5.8.2. En plus de permettre la détection de débordements de tampons, Splint permet de détecter des vulnérabilités de chaîne de format, et plusieurs «abus» du langage C. L'analyse est se base sur des annotations que le programmeur doit souvent ajouter au déclaration des fonctions et elle est effectuée au niveau d'une fonction à la fois. Bien que certaines erreurs puissent lui échapper, la plupart des aspects du langage C sont tenu en compte par Splint. Ce programme s'appelait anciennement LCLint. `http://www.splint.org/`

RATS (Libre) Nous avons discuté de ce programme à la section 5.8.1. Il effectue une analyse lexicale qui permet de détecter des constructions «dangereuses» dans les langages C, C++ et dans certains autres langages. Cette approche lui permet de trouver des débordements de tampons,

des vulnérabilités de chaîne de format et d'autres problèmes potentiels de sécurité. http://www.securesoftware.com/download_form_rats.htm

Flawfinder (Libre) Nous avons discuté de ce programme à la section 5.8.1. Il effectue une analyse lexicale qui permet de détecter des constructions «dangereuses» dans les langages C et C++. Cette approche lui permet de trouver des débordements de tampons, des vulnérabilités de chaîne de format et d'autres problèmes potentiels de sécurité. http://www.dwheeler.com/flawfinder/

ITS4 (Gratuit) Nous avons discuté de ce programme à la section 5.8.1. Il effectue une analyse lexicale qui permet de détecter des constructions «dangereuses» dans les langages C et C++. Cette approche lui permet de trouver des débordements de tampons et d'autres problèmes potentiels de sécurité. http://www.cigital.com/its4/

BOON (Partiellement libre) Nous avons discuté du fonctionnement de ce programme à la section 5.8.3. Il permet de détecter des débordements de tampons en considérant les chaînes de caractères comme un type abstrait manipulé par les fonctions de la bibliothèque C. L'analyse effectuée est interprocédurale, mais pas très précise. La détection des débordements est effectuée par la résolution d'un système de contrainte sur des intervalles d'entiers représentant l'espace allouée pour les chaînes de caractères et leur longueur. Les débordements qui sont effectués autrement que par l'appel d'une fonction de la bibliothèque C sont complètement ignorés. http://www.cs.berkeley.edu/~daw/boon/

Wasp (Gratuit) Ce programme est décrit dans [Mat01]. Il permet de détecter des débordements de tampons et il semble effectuer une analyse interprocédurale. http://www.waspsoft.com/

Cqual (Libre) Il effectue une analyse basée sur le typage et permet de spécifier et de vérifier des propriétés de programmes C. Il demande à l'utilisateur d'ajouter certaines annotations à quelques endroits clés dans le code source du programme à analyser. Une de ces applications est de permettre la détection de vulnérabilité de chaîne de format [Fos02]. http://www.cs.berkeley.edu/~jfoster/cqual/

PScan (Libre) Il effectue une analyse lexicale qui permet de détecter des utilisations «dangereuses» de fonctions prenant en paramètre une chaîne de format [DeK00]. http://www.striker.ottawa.on.ca/~aland/pscan/

B.2 Dialectes du C

Cyclone (Libre) C'est un langage de programmation basé sur le C qui ne permet pas les débordements de tampons, ni les vulnérabilités de chaîne de format. Il a été mentionné brièvement à la section 5.2. http://www.research.att.com/projects/cyclone/

B.3 Vérification des bornes à l'exécution

GCC — Bounded Pointers (Libre) C'est une extension à GCC qui modifie la représentation des pointeurs de manière à permettre efficacement la vérification des bornes des tableaux à l'exécution. La technique utilisée est décrite à la section 5.5.4.1. `http://gcc.gnu.org/projects/bp/main.html`

Bounds Checking for C (Libre) C'est une extension de GCC qui permet la vérification des bornes à l'exécution sur la plupart des pointeurs sans en modifier la représentation. La technique utilisée est décrite à la section 5.5.4.3. `http://web.inter.nl.net/hcc/Haj.Ten.Brugge/` et `ftp://nscs.fast.net/pub/binaries/boundschecking/`

CCured (Libre) Ce programme analyse du code source C pour ajouter le nombre minimal de vérifications à l'exécution de manière à éviter les débordements de tampons. Ce qu'il génère en sortie est du code C qui doit à son tour être compilé [ccu02]. `http://manju.cs.berkeley.edu/ccured/`

StackGuard (Libre) C'est une extension à GCC qui permet de protéger l'adresse de retour des fonctions sur la pile. Les techniques utilisées sont l'utilisation d'un canari et la protection assistée par le processeur. Elles sont décrites aux sections 5.5.1.1 et 5.5.1.2. Selon les versions et la configuration, le canari utilisé est soit nul, de terminaison ou aléatoire. La version 1.21 supportait également un canari appelé «OU Exclusif» qui permettait d'éviter certaines attaques supplémentaires, mais cette variante ne semble plus disponible dans la version 2.0. La protection des adresses de retour assistée par le processeur (MemGuard) n'était disponible que dans les toutes premières versions parce que l'approche ne s'est pas avérée assez performante. StackGuard n'effectue donc pas une véritable vérification des bornes et il n'empêche aucun débordement. Il empêche seulement la réussite de certaines attaques. Pour ce faire, le code généré par le compilateur pour le prologue et l'épilogue des fonctions est altéré. `http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/` et `http://www.immunix.org/stackguard.html`

Stack Shield (Libre) Il permet de protéger l'adresse de retour des fonctions sur la pile. La technique employée est l'utilisation d'une pile alternative. Elle est décrite à la section 5.5.2. La pile alternativei n'est cependant pas illimitée et par défaut elle comporte 256 éléments. Ça signifie que si plus de 256 appels de fonctions sont imbriqués, les derniers appels ne bénéficient pas de la protection. Stack Shield fonctionne en modifiant le code assembleur généré lors de la compilation par GCC. Il n'effectue pas de vérification des bornes et il n'empêche aucun débordement. Il empêche seulement certaines attaques de se produire. Ce logiciel ne semble plus être en développement. `http://www.angelfire.com/sk/stackshield/`

Stack-Smashing Protector (SSP) (Libre) C'est une extension à GCC qui permet de protéger l'adresses de retour et plusieurs autres données «importantes» qui se trouvent sur la pile. Pour ce faire il utilise des canaris et il réordonne les variables de la pile, des techniques décrites respectivement aux sections 5.5.1.1 et 5.5.3. Il n'effectue pas de vérification des bornes, et il n'empêche aucun débordement, mais il permet d'éviter plusieurs attaques. Il fonctionne en modifiant le code au niveau du langage intermédiaire généré par GCC. Il évite d'ajouter la vérification de la valeur

du canari ou la copie des paramètres s'il peut déterminer que le programme s'exécutera toujours correctement sans ces ajouts. Par contre, l'optimiseur de GCC peut enlever certaines vérifications ajoutées par SSP si le niveau d'optimisation demandé est trop élevé. Ce projet était anciennement connu sous le nom Propolice. `http://www.trl.ibm.com/projects/security/ssp/`

libsafe (Libre) C'est une bibliothèque de remplacement pour certaines fonctions de la bibliothèque C. Elle fonctionne sous Linux et elle permet d'éviter l'écrasement de l'adresse de retour des fonctions protégées ainsi que les vulnérabilités de chaîne de format. Ces techniques sont décrites aux sections 5.7.1.2 et 5.7.1.3. Cette bibliothèque n'empêche pas tous les débordements, mais elle empêche ceux qui sont les moins subtils. `http://www.research.avayalabs.com/project/libsafe/`

libparanoia (Libre) C'est une bibliothèque de remplacement pour certaines fonctions de la bibliothèque C. Elle fonctionne sous FreeBSD et elle permet d'éviter l'écrasement de l'adresse de retour des fonctions protégées. Cette technique est décrite à la section 5.7.1.2. Cette bibliothèque n'empêche pas tous les débordements, mais elle empêche ceux qui sont les moins subtils. `http://www.lexa.ru/snar/libparanoia/`

strlcpy() et strlcat() (Libre) Ces fonctions offrent des interfaces alternatives, moins enclines aux débordements de tampons, pour le traitement de chaînes de caractères. Elles sont décrites à la section 5.7.1.5.

libmib — astring (Gratuit) C'est une bibliothèque alternative qui gère l'allocation et la réallocation des chaînes de caractères de manière à éviter les débordements. La technique utilisée est décrite à la section 5.7.1.5. `http://www.mibsoftware.com/libmib/astring/`

Checker (Libre) Ce programme est un outil de débogage qui permet de trouver à l'exécution les lectures et les écritures à l'extérieur de la mémoire allouée et la lecture de mémoire qui n'est pas initialisée. Il permet donc de trouver certains débordements de tampons. La technique qu'il utilise est décrite à la section 5.5.4.4. Il permet aussi de trouver des blocs de mémoire alloués qui ne sont plus référencés alors qu'ils n'ont pas été libérés. Le programme à vérifier doit être instrumenté à l'aide d'une extension au compilateur GCC. `http://www.gnu.org/software/checker/checker.html`

Valgrind (Libre) Ce programme est un outil de débogage qui permet de trouver à l'exécution les lectures et les écritures à l'extérieur de la mémoire allouée et la lecture de mémoire qui n'est pas initialisée. Il permet donc de trouver certains débordements de tampons. La technique qu'il utilise est décrite à la section 5.5.4.4. Il permet même de suivre l'utilisation de la mémoire bit par bit. Il permet aussi de trouver des blocs de mémoire alloués qui ne sont plus référencés alors qu'ils n'ont pas été libérés et d'autres types d'erreurs. Il permet de plus d'effectuer une simulation de la mémoire cache pour profiler son utilisation. Un compilateur JIT permet d'instrumenter directement le code machine du programme à vérifier (IA-32 vers IA-32). `http://developer.kde.org/~sewardj/`

Rational Purify (Commercial) Ce programme est un outil de débogage qui permet de trouver à l'exécution les lectures et les écritures à l'extérieur de la mémoire allouée et la lecture de mémoire qui n'est pas initialisée. Il permet donc de trouver certains débordements de tampons. La technique qu'il utilise est décrite à la section 5.5.4.4. Il permet aussi de trouver des blocs de mémoire alloués qui ne sont plus référencés alors qu'ils n'ont pas été libérés. Le code machine du programme à vérifier est modifié dynamiquement lors de l'exécution pour le suivi des accès à la mémoire. Ce logiciel n'est pas disponible sous Linux. `http://www.rational.com/products/purify_unix/index.jsp`

Electric Fence (Libre) C'est une bibliothèque de remplacement pour les fonctions de gestion de la mémoire. Elle alloue la mémoire de façon à ce que les accès après la zone allouée (ou avant) génèrent une faute. Cette technique est décrite à la section 5.7.1.1. `http://perens.com/FreeSoftware/`

B.4 Vérification des vulnérabilités de chaîne de format à l'exécution

libsafe (Libre) Voir la section B.3.

FormatGuard (Libre) C'est une extension à glibc qui permet de s'assurer que le bon nombre de paramètres est passé à certaines fonctions de la bibliothèque C qui reçoivent en paramètre une chaîne de format. La technique utilisée par ce programme est décrite à la section 5.7.1.4. `http://www.immunix.org/formatguard.html`

B.5 Outils de test

Fuzz (Gratuit) Ce programme donne des entrées aléatoires à un programme dans le but de trouver des problèmes. Il a permis de détecter de nombreux débordements de tampons. Nous l'avons mentionné à la section 5.3. `http://www.cs.wisc.edu/~bart/fuzz/fuzz.html`

BFBTester (Libre) Cet outil vérifie la présence de débordement dans l'utilisation des arguments et des variables d'environnement passés à un programme. `http://bfbtester.sourceforge.net/`

B.6 Autres vérifications dynamiques

Les programmes de cette section n'empêchent pas directement les débordements de tampons ni les vulnérabilités de chaîne de format, mais ils peuvent grandement limiter leurs conséquences.

Linux kernel patch from Openwall Project (Libre) Cette retouche au noyau Linux empêche l'exécution de code sur la pile et elle modifie l'adresse utilisée pour le chargement des bibliothèques liées dynamiquement. Ces techniques sont décrites aux sections 5.6.2 et 5.7.2.1. Elle empêche également d'autres actions souvent utilisées par les pirates. `http://www.openwall.com/linux/`

PaX (Libre) Cette extension du noyau Linux permet de rendre exécutable ou non chaque page de l'espace d'adressage d'un programme. Il place aussi les bibliothèques partagées ainsi que la pile à des adresses aléatoires de manière à ce qu'il soit plus difficile de trouver la bonne adresse à utiliser lors d'une attaque. Ces techniques sont décrites dans la première partie de la section 5.6.3 et aux sections 5.7.2.2 et 5.7.3. `http://pageexec.virtualave.net/`

kNoX (Libre) Cette retouche au noyau Linux 2.2 empêche l'exécution de code sur toutes les pages accessibles en écriture et elle modifie l'adresse utilisée pour le chargement des bibliothèques liées dynamiquement. Ces techniques sont décrites à la deuxième partie de la section 5.6.3 et à la section 5.7.2.1. Elle empêche également d'autres actions souvent utilisées par les pirates. `http://isec.pl/projects/knox/knox.html`

RSX (Libre) Cette extension du noyau Linux 2.4 permet de rendre exécutables ou non certaines pages de l'espace d'adressage d'un programme. La technique qu'il utilise est décrite à la deuxième partie de la section 5.6.3. `http://www.starzetz.com/software/rsx/`

syscalltrack (Libre) Ce logiciel est en grande partie une extension au noyau Linux qui permet entre autres de filtrer les appels au noyau effectués par tous les programmes qui s'exécutent sur un système. Il a été mentionné brièvement à la section 5.4. `http://syscalltrack.sourceforge.net/`

Janus (Libre) Ce logiciel permet d'exécuter un programme dans un environnement contrôlé, c'est-à-dire où ses actions sont scrutées et vérifiées pour qu'elles soient conformes à une politique de sécurité. Il a été mentionné brièvement à la section 5.4. `http://www.cs.berkeley.edu/~daw/janus/`

SUBTERFUGUE (Libre) C'est un cadre d'application qui permet de jouer avec la réalité des programmes. Il permet de scruter les appels au noyau effectués par un programme pour voir ce qu'il tente de faire ou l'empêcher d'effectuer certaines actions. Il a été mentionné brièvement à la section 5.4. `http://subterfugue.org/`

Bibliographie

[ABS93] Todd M. Austin, Scott E. Breach et Gurindar S. Sohi. Efficient detection of all pointer and array access errors. Rapport technique 1197, Computer Sciences Department University of Wisconsin, 1993.

[Ano02] Anonymous author. Building ptrace injecting shellcodes. *Phrack*, 11(59), July 2002. URL `http://www.phrack.com/phrack/59/p59-0x0c.txt`.

[Ava] Avaya Labs, Avaya Inc. *Libsafe manual*. URL `http://www.research.avayalabs.com/project/libsafe/doc/libsafe.8.html`.

[Bea01] Maxime Beaudoin. Données malicieuses : théorie et analyse. Rapport technique DIUL-RR-0105, Département d'informatique, Université Laval, August 2001.

[BK00] Bulba et Kil3r. Bypassing StackGuard and StackShield. *Phrack*, 10(56), November 2000. URL `http://www.phrack.com/phrack/56/p56-0x05`.

[Bou00a] Pascal Bouchareine. __atexit in memory bugs. Vuln-Dev mailing list, December 2000. URL `http://archives.neohapsis.com/archives/vuln-dev/2000-q4/att-0665/01-heap_atexit.txt`.

[Bou00b] Pascal Bouchareine. Format string vulnerability. Rapport technique, Hacker Emergency Response Team, July 2000. URL `http://www.hert.org/papers/format.html`.

[BTS99] Arash Baratloo, Timothy Tsai et Navjot Singh. Libsafe : Protecting critical elements of stacks. Rapport technique, Bell Labs, Lucent Technologies, December 1999. URL `http://www.research.avayalabs.com/project/libsafe/doc/libsafe.pdf`.

[Cav] Forrest J. Cavalier III. Libmib allocated string functions. URL `http://mibsoftware.com/libmib/astring/`.

[CBB+01] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen et Jamie Lokier. Formatguard : Automatic protection from printf format string vulnerabilities. Rapport technique, WireX Communications, Inc., Purdue University, CERN, August 2001. URL `http://www.immunix.org/formatguard.pdf`.

[ccu02] Ccured documentation, 2002. URL `http://manju.cs.berkeley.edu/ccured/`.

[Ces00] Silvio Cesare. ELF executable reconstruction from a core image, January 2000. URL `http://vx.netlux.org/lib/vsc03.html`.

[Con99] Matt Conover. w00w00 on heap overflows. Rapport technique, w00w00 Security Team, January 1999. URL `http://www.w00w00.org/articles.html`.

[Cow00] Crispin Cowan. Stackguard 1.21 vulnerability. BugTraq mailing list, August 2000. URL http://security-archive.merton.ox.ac.uk/bugtraq-200008/0301.html.

[CPM+98] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle et Qian Zhang. StackGuard : Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78. San Antonio, Texas, jan 1998. URL http://www.immunix.org/StackGuard/usenixsc98.pdf.

[CWP+99] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie et Jonathan Walpole. Buffer overflows : Attacks and defenses for the vulnerability of the decade. Rapport technique, Oregon Graduate Institute of Science & Technology, 1999. URL http://www.immunix.org/StackGuard/discex00.pdf.

[cyc02] Cyclone, 2002. URL http://www.research.att.com/projects/cyclone/.

[DeK00] Alan DeKok. Pscan : A limited problem scanner for C source files, July 2000. URL http://www.striker.ottawa.on.ca/~aland/pscan/.

[Des97a] Solar Designer. Getting around non-executable stack (and fix). BugTraq mailing list, August 1997. URL http://clip.dia.fi.upm.es/~alopez/bugs/bugtraq2/0287.html.

[Des97b] Solar Designer. Non-executable stack – final linux kernel patch, May 1997. URL http://www.uwsg.iu.edu/hypermail/linux/kernel/9705.1/0493.html.

[Des02] Solar Designer. Linux kernel patch from the Openwall Project (README and FAQ), September 2002. URL http://www.openwall.com/linux/.

[Dup02] Kasper Dupont. Why a stack with exec flag ? comp.os.linux.security group, June 2002. URL http://cert.uni-stuttgart.de/archive/usenet/comp.os.linux.security/2002/06/msg00292.html.

[EL02] David Evans et David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1) :42–51, Jan/Feb 2002. URL http://www.cs.virginia.edu/~evans/pubs/ieeesoftware.pdf.

[EY00] Hiroaki Etoh et Kunikazu Yoda. Protecting from stack-smashing attacks. Rapport technique, IBM Research Division, Tokyo Research Laboratory, June 2000. URL http://www.trl.ibm.com/projects/security/ssp/main.html.

[Foo02a] Foon. Shatter attacks — how to break Windows, August 2002. URL http://security.tombom.co.uk/shatter.html.

[Foo02b] Foon. Shatter attacks — more techniques, more detail, more juicy goodness, August 2002. URL http://security.tombom.co.uk/moreshatter.html.

[Fos02] Jeff Foster. Cqual : A tool for adding type qualifiers to C, June 2002. URL http://www.cs.berkeley.edu/~jfoster/cqual/.

[Fry00] Niklas Frykholm. Countermeasures against buffer overflow attacks. Rapport technique, RSA Security, November 2000. URL http://www.rsasecurity.com/rsalabs/technotes/buffer/buffer_overflow.html.

[GBPdlHQ+02] Jesús M. González-Barahona, Miguel A. Ortuño Pérez, Pedro de las Heras Quirós, José Centeno González et Vicente Matellán Olivera. Counting potatoes : The size of Debian 2.2, December 2002. URL http://people.debian.org/~jgb/debian-counting/.

[Gin98] Tristan Gingold. *Checker*, 1998. URL `ftp://alpha.gnu.org/gnu/checker/Checker-0.9.9.1.tar.gz`.

[HJ91] Reed Hastings et Bob Joyce. Purify : Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–136. USENIX Association, USENIX Association, 1991.

[Imm97] Immunix. Immunix canary patch to GCC 2.7.2.2 — a buffer overflow exploit detector. Rapport technique, Immunix, December 1997. URL `ftp://ftp.cse.ogi.edu/pub/dsrg/immunix/stackguard-gcc.README`.

[Imm00] Immunix. StackGuard mechanism : Emsi's vulnerability. Rapport technique, Immunix, 2000. URL `http://immunix.org/StackGuard/emsi_vuln.html`.

[Int00] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3 : System Programming Guide*, 2000. URL `http://www.intel.com/design/pentium4/manuals/24547209.pdf`.

[jan00] Janus, 2000. URL `http://www.cs.berkeley.edu/~daw/janus/`.

[JK97] Richard W. M. Jones et Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. Rapport technique, Department of Computing Imperial College of Science, Technology and Medicine, 1997. URL `http://www.doc.ic.ac.uk/~phjk/Publications/BoundsCheckingForC.ps.gz`.

[Jon95] Richard W. M. Jones. A bounds checking C compiler. Rapport technique, Department of Computing Imperial College of Science, Technology and Medicine, May 1995. URL `ftp://nscs.fast.net/pub/binaries/boundschecking/bounds-checking-reports.tar.bz2`.

[Ket98] Richard Kettlewell. Protecting against some buffer-overrun attacks, April 1998. URL `http://www.greenend.org.uk/rjk/random-stack.html`.

[klo99] klog. The frame pointer overwrite. *Phrack*, 9(55), September 1999. URL `http://www.phrack.com/phrack/55/P55-08`.

[KS02] Paul A. Karger et Roger R. Schell. Thirty years later : Lessons from the Multics security evaluation. Rapport technique, IBM Research, July 2002.

[LE01] David Larochelle et David Evans. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Security Symposium*, August 2001. URL `http://lclint.cs.virginia.edu/usenix01.pdf`.

[Mat01] Vincent Mathieu. Outils d'analyse statique. Rapport technique DIUL-RR-0106, Département d'informatique, Université Laval, August 2001.

[McG98a] Greg McGary. Bounds checking. egcs mailing list, May 1998. URL `http://ftp.azc.uam.mx/mirrors/gnu/egcs/mail-archives/1998/egcs-1998-05.bz2`.

[McG98b] Greg McGary. Technical specification for bounded pointers in GCC. egcs mailing list (Re : array bounds checking ?), May 1998. URL `http://ftp.azc.uam.mx/mirrors/gnu/egcs/mail-archives/1998/egcs-1998-05.bz2`.

[McG99] Greg McGary. Support array bounds checking. egcs-patches mailing list, June 1999. URL `http://ftp.azc.uam.mx/mirrors/gnu/egcs/mail-archives/1999/egcs-patches-1999-06.bz2`.

[MdR99] Todd C. Miller et Theo de Raadt. strlcpy and strlcat — consistent, safe, string copy and concatenation. In *Proceedings of the FREENIX Track : 1999 USENIX*

Annual Technical Conference. USENIX, USENIX, June 1999. URL http://www.usenix.org/events/usenix99/full_papers/millert/millert.pdf.

[Mic02] Microsoft Corporation. *Information About Reported Architectural Flaw in Windows*, September 2002. URL http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/topics/htshat.asp.

[Mil00] Todd C. Miller. *strlcpy manual*, 2000. URL ftp://ftp.openbsd.org/pub/OpenBSD/src/lib/libc/string/strlcpy.3.

[MKL+95] Barton P. Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan et Jeff Steidl. Fuzz revisited : A re-examination of the reliability of UNIX utilities and services. Rapport technique, University of Wisconsin, 1995. URL ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz-revisited.pdf.

[MV00] Gary McGraw et John Viega. Make your software behave : Preventing buffer overflows. *developerWorks*, March 2000. URL http://www-106.ibm.com/developerworks/security/library/s-buffer-defend.html.

[New00] Tim Newsham. Format string attacks. Rapport technique, Guardent, Inc, September 2000. URL http://julianor.tripod.com/tn-usfs.pdf.

[One] Aleph One. Bugtraq frequently asked questions. URL http://www.nationwide.net/~aleph1/FAQ.

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996. URL http://www.phrack.org/phrack/49/P49-14.

[Per93] Bruce Perens. *Electric Fence manual (libefence.3)*, 1993. URL ftp://ftp.perens.com/pub/ElectricFence/.

[por01] portal. Format string exploitation demystified, March 2001.

[Rit93] Dennis M. Ritchie. The development of the C language. Rapport technique, Bell Labs/Lucent Technologies, Murray Hill NJ 07974 USA, 1993. URL http://cm.bell-labs.com/cm/cs/who/dmr/chist.pdf.

[rix00] rix. Smashing C++ vptrs. *Phrack*, 10(56), May 2000. URL http://www.phrack.com/phrack/56/P56-0x08.

[Rob01] Tim J. Robbins. libformat - protection against format string attacks, 2001. URL http://box3n.gumbynet.org/~fyre/software/libformat.html.

[RWM02] Chris Ren, Michael Weber et Gary McGraw. Microsoft compiler flaw technical note. Rapport technique, Cigital, 2002. URL http://www.cigital.com/news/mscompiler-tech.html.

[scu01] scut. Exploiting format string vulnerabilities. Rapport technique, Team teso, September 2001. URL http://www.team-teso.net/articles/formatstring/.

[Sec02] Secure Software. Rats readme, 2002. URL http://www.securesoftware.com/download_form_rats.htm.

[Smi97] Nathan P. Smith. Stack smashing vulnerabilities in the UNIX operating system. Rapport technique, Southern Connecticut State University, Computer Science Department, May 1997. URL http://millcomm.com/~nate/machines/security/stack-smashing/.

[Sna97] Alexander Snarskii. Increasing overall security...., February 1997. URL ftp://ftp.lucky.net/pub/unix/local/libc-letter.

[Sna00] Alexandre Snarskii. libparanoia, 2000. URL http://www.lexa.ru/snar/libparanoia/.

[Sta01] Paul Starzetz. Announcing RSX - non exec stack/heap module. BugTraq mailing list, June 2001. URL http://cert.uni-stuttgart.de/archive/bugtraq/2001/06/msg00060.html.

[sub02] Subterfugue, 2002. URL http://subterfugue.org/.

[sys02] syscalltrack, 2002. URL http://syscalltrack.sourceforge.net/.

[sz00] scut et z. Teso wu-ftpd 2.6.0 exploit : 7350wu, December 2000. URL http://www.team-teso.net/releases.php.

[Tea00] PaX Team. PaX design & implementation. Rapport technique, The PaX Team, November 2000. URL http://pageexec.virtualave.net/pageexec.txt.

[Tea02] PaX Team. Why a stack with exec flag ? comp.os.linux.security group, June 2002. URL http://cert.uni-stuttgart.de/archive/usenet/comp.os.linux.security/2002/06/msg00378.html.

[Tec02] CORE Security Technologies. Multiple vulnerabilities in stack smashing protection technologies. Rapport technique, CORE Security Technologies, April 2002. URL http://www.core-sdi.com/common/showdoc.php?idx=221\&idxseccion=10.

[tf800] tf8. Wuftpd : Providing *remote* root since at least 1994. BugTraq mailing list, June 2000. URL http://online.securityfocus.com/archive/1/66367.

[Thu01] Andreas Thuemmel. Analysis of format string bugs, March 2001. URL http://www.securityfocus.com/data/library/format-bug-analysis.pdf.

[TS01] Timothy Tsai et Navjot Singh. Libsafe 2.0 : Detection of format string vulnerability exploits. Rapport technique, Avaya Labs, Avaya Inc., February 2001. URL http://www.research.avayalabs.com/project/libsafe/doc/whitepaper-20.pdf.

[Twi99a] Tymm Twillman. Exploit for proftpd 1.2.0pre6. BugTraq mailing list, September 1999. URL http://online.securityfocus.com/archive/1/28143.

[Twi99b] Tymm Twillman. proftpd 1.2.0pre6 patch. BugTraq mailing list, September 1999. URL http://online.securityfocus.com/archive/1/27744.

[VBKM01] John Viega, J.T. Bloch, Tadayoshi Kohno et Gary McGraw. ITS4 : A static vulnerability scanner for C and C++ code. Rapport technique, Reliable Software Technologies, December 2001. URL http://www.cigital.com/papers/download/its4.pdf.

[Ven00a] Vendicator. Stack Shield README file v0.7, 2000. URL http://www.angelfire.com/sk/stackshield/download.html.

[Ven00b] Vendicator. Stack Shield tecnical info file v0.7, 2000. URL http://www.angelfire.com/sk/stackshield/download.html.

[Wag00] David A. Wagner. *Static analysis and computer security : New techniques for software assurance*. Thèse de doctorat, University of California at Berkley, 2000. URL http://www.cs.berkeley.edu/~daw/papers/phd-dis.ps.

[WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer et Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17. San Diego, CA, February 2000. URL `http://www.cs.berkeley.edu/~daw/papers/overruns-ndss00.pdf`.

[Whe01] David A. Wheeler. More than a gigabuck : Estimating GNU/Linux's size, July 2001. URL `http://www.dwheeler.com/sloc/`.

[Whe02a] David A. Wheeler. *FlawFinder manual*, 2002. URL `http://www.dwheeler.com/flawfinder/flawfinder.pdf`.

[Whe02b] David A. Wheeler. *FlawFinder source code*, 2002. URL `http://www.dwheeler.com/flawfinder/`.

[Whe02c] David A. Wheeler. *Secure Programming for Linux and Unix HOWTO*. www.dwheeler.com, 2002. URL `http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO.pdf`.

[Woj98] Rafal Wojtczuk. Defeating Solar Designer non-executable stack patch. BugTraq mailing list, January 1998. URL `http://bugtraq.inet-one.com/dir.1998-01/msg00151.html`.

[Woj01] Rafal Wojtczuk. The advanced return-into-lib(c) exploits : PaX case study. *Phrack*, 11(58), December 2001. URL `http://www.phrack.com/phrack/58/p58-0x04`.

[WU-a] WU-FTPD Development Group. Patch for wu-ftpd 2.6.0 (lreply). URL `ftp://ftp.wu-ftpd.org/pub/wu-ftpd/patches/apply_to_2.6.0/lreply-buffer-overflow.patch`.

[WU-b] WU-FTPD Development Group. Wu-ftpd 2.6.0. URL `ftp://ftp.wu-ftpd.org/pub/wu-ftpd-attic/wu-ftpd-2.6.0.tar.gz`.

Index