Christophe CARMAGNAC
Romain THEODET

Les buffer overflows : Qu’est-ce, et comment s’en protéger

0 - Qu’est-ce qu’un buffer overflow

Un buffer overflow est une faille dans laquelle un attaquant exploite un buffer non protégé, afin d’écrire dans des zones de mémoires qu’il n’est pas censé pouvoir modifier. Le but est de pouvoir exécuter du code arbitraire sur la machine de la victime, et de ne pas respecter le fonctionnement normal de l’application.

Le buffer overflow de base se contente de récrire tout le code de la fonction jusqu’à arriver à l’adresse de retour de la fonction. Cette adresse de retour est une valeur utilisée pour déterminer l’endroit où se trouve la prochaine instruction à exécuter une fois que la fonction est terminée. L’objectif de l’attaquant est donc de pouvoir la modifier afin d’exécuter le code qu’il a inséré dans son buffer.

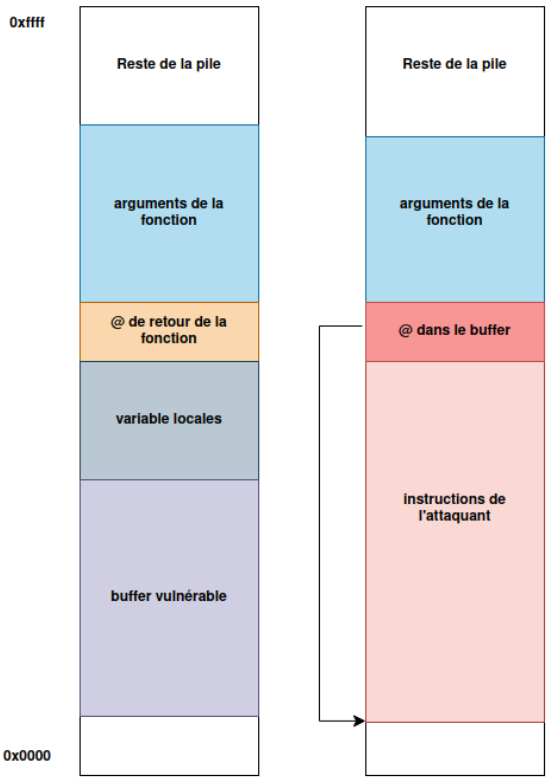
Un langage vulnérable à ce type d’attaques respecte plusieurs critères :

- La présence de buffers de taille variable (tableaux, chaînes de caractères…)
- Une absence de vérification systématique des indices, permettant de lire en dehors desdits buffers

Dans des langages bas niveau, on distingue plusieurs types de mémoire :

- Le tas, ou heap, qui est allouée dynamiquement et stocke des données
- La pile, ou stack, qui stocke aussi des données, mais de façon temporaire et locale à la fonction
- Le texte, ou “code”, composé cette fois des différentes instructions à exécuter

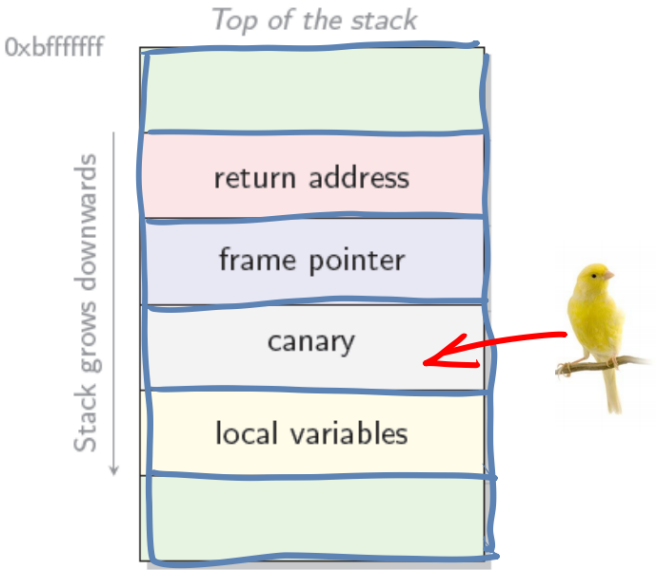
Voici un schéma visualisant la pile avant et après un buffer overflow.



Ce schéma illustre parfaitement un buffer overflow classique ; la mémoire est écrasée jusqu'à l'adresse de retour située dans la stack frame. Une fois la fonction terminée, lorsque le programme essaiera de retourner dans la fonction parente, il retournera en réalité là où l'attaquant l'a désiré.

1 - Les canaris

Un canari est une valeur insérée après l'adresse de retour de la fonction. Une fois la fonction finie, le système d'exploitation va vérifier si cette valeur n'a pas été modifiée. Si c'est le cas, alors le système d'exploitation mettra en général fin à l'exécution du programme, constatant une violation de la mémoire.



Historiquement, ce terme provient à l'origine du monde ouvrier, où un canari en chair et en os prévenait aux mineurs les coups de grisou. Tant que le canari était en vie, l'activité pouvait poursuivre ; néanmoins, un brutal changement de comportement de l'animal alertait naturellement les mineurs d'une possible fuite de gaz.

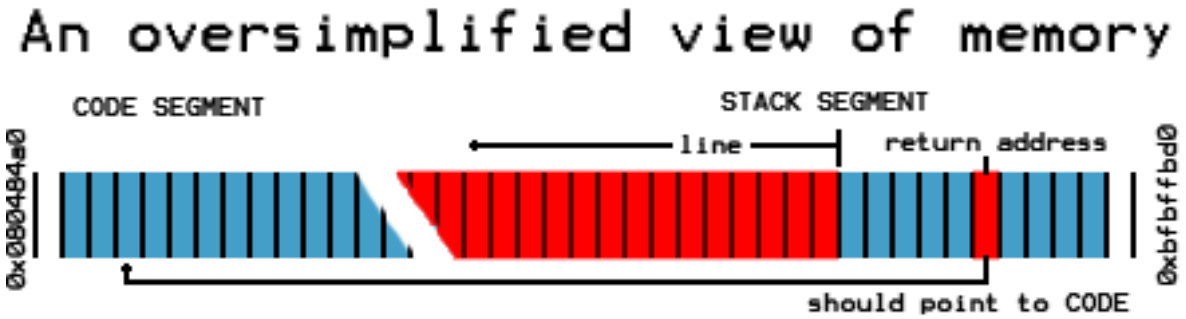
À première vue, il semblerait logique qu'une implémentation des canaries puisse générer des valeurs aléatoires dynamiquement. Cette approche apparaît cependant très coûteuse, et un tradeoff s'impose donc. L'implémentation classique est donc de générer le canari à la compilation, ce qui diminue son efficacité, mais réduit drastiquement l'impact en performances.

La meilleure solution actuellement est de laisser le compilateur décider par lui-même quelles fonctions protéger du fait de leur criticité. L'impact est donc assez négligeable, d'environ 0.5% en termes de temps d'exécution, et pour un binaire très légèrement plus large (~1%). Cette mesure est gérée à un niveau logiciel, et peut donc être implémentée différemment selon les compilateurs et les systèmes d'exploitations.

Il existe plusieurs méthodes pour contourner un canari. Soit l'attaquant essaie de trouver des valeurs du canari en essayant des valeurs aléatoires jusqu'à qu'il trouve. Soit sinon l'attaquant peut essayer d'utiliser la fonction `printf` pour afficher le canari. Soit enfin, l'attaquant peut, s'il en a la possibilité, abuser de la fonction `memcpy` afin d'écrire dans des endroits arbitraires de la mémoire et ainsi sauter le canari sans problème. Cette dernière méthode dépasse cependant le cadre d'un simple buffer overflow. Enfin, notons aussi que les canaris étant déterminés à la compilation, cette méthode est assez peu efficace si le binaire est connu de l'attaquant, puisque alors le canari pourra facilement être extrait en décompilant l'exécutable.

2 - ESP

ESP, acronyme d'*Executable Space Protection*, est une protection permettant d'éviter qu'une plage de données soit à la fois exécutable et modifiable. Cela permet en théorie d'éviter l'exécution des instructions mises dans la pile par l'attaquant.



L'implémentation de cette technique se fait généralement au niveau matériel, via l'activation d'un bit spécial, le NX bit. Elle évite le vecteur d'attaque suivant :

```c
#include <stdlib.h>
#include "stdio.h"

int main(int argc, char **argv) {
    printf( format: "Quel âge avez-vous ?\n");
    char ageStr[3];
    scanf( format: "%s", ageStr);

    int age = atoi( nptr: ageStr);
    if (age < 18) {
        printf( format: "Vous êtes mineur.\n");
    } else {
        printf( format: "Vous êtes majeur.\n");
    }
}
```

```
Quel âge avez-vous ?
21
push ax
push bx
mov ah, 0x0e
.loop:
cmp [bx], byte 0
je .exit
mov al, [bx]
int 0x10
inc bx
jmp .loop
.exit:
pop ax
pop bx
ret
```

Dans cette attaque, simplifiée à un but pédagogique, l'attaquant effectue un buffer overflow en insérant du code (en réalité sous format binaire compilé), pouvant ainsi exécuter du code arbitraire.

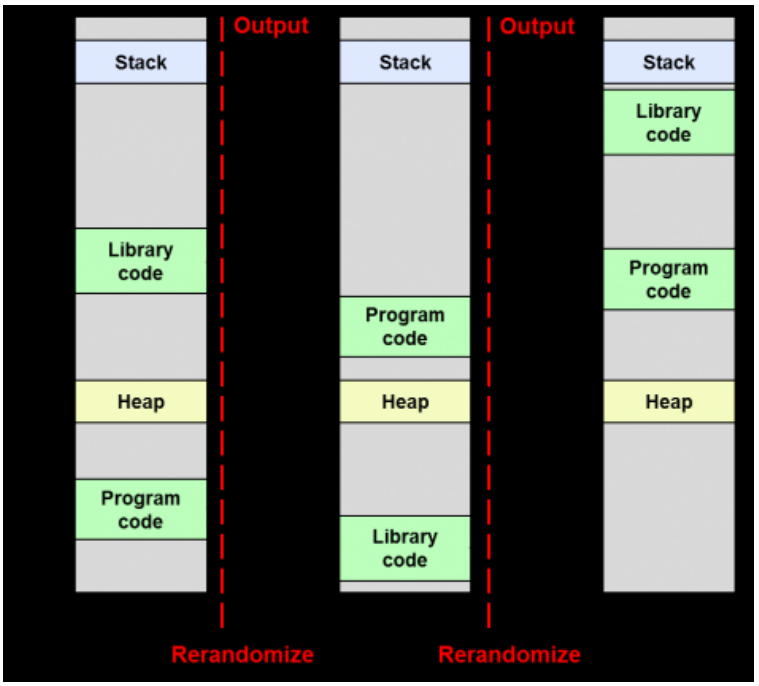
Cette protection peut là encore être défaite en utilisant du code préexistant. L'idée étant de réécrire l'adresse de retour de la fonction pour retourner dans du code préexistant, soit écrit par le développeur lui-même, soit du code présent dans une bibliothèque telle que la librairie C standard. En enchaînant ces retours les uns à la suite des autres, on peut alors se débrouiller pour exécuter du code arbitraire, la librairie C standard contenant de nombreuses fonctions dangereuses telles que `system`, permettant l'exécution de commandes bash.

Les impacts d'ESP sur les performances d'exécution du code sont nuls, du fait de l'implémentation matérielle de la fonctionnalité.

3 - ASLR

ASLR, pour *Address Space Layout Randomization*, consiste à rendre aléatoire l'emplacement des segments de code et de données dans la mémoire.

Cet adressage requiert un système d'adressage virtuel, permettant aux applications de référencer les fonctions tout en évitant à l'attaquant de trouver leur position. Un compromis temps / mémoire est alors nécessaire entre rendre aléatoire l'espace mémoire dynamiquement ou en début d'exécution de programme, ce qui explique une implémentation très différente de cet adressage entre les différents systèmes d'exploitation



Les attaques sont là encore toujours possibles, bien que bien plus complexes. Un espace 32 bits n'est pas assez grand pour résister à une attaque via brute force, contrairement à un adressage sur 64 bits, bien que l'utilisation de NOP-slides pour augmenter la probabilité de réussir l'attaque est possible. Le but d'une NOP-slide est d'augmenter la probabilité de réussite d'une attaque. L'instruction NOP, ou No-Op pour *No-Operation*, indique au processeur qu'il doit passer à l'instruction suivante. Cette commande est utile pour gâcher des cycles processeurs et attendre sans rien changer dans la mémoire. En écrivant sur une grosse portion de mémoire des instructions NOP, l'attaquant peut plus facilement retrouver les instructions qu'il a mises dans son buffer.

Les impacts de cette défense sur l'exécution des programmes dépendent des systèmes d'exploitations. Windows limitera par exemple la mémoire maximale utilisable, alors que Linux prendra plus de temps à s'exécuter, entre 5 et 10%. Cet impact performance se réduit cependant pour un OS 64 bits, et n'est donc un réel problème que pour certains binaires 32-bits.

Conclusion

Alors, au vu des précédentes protections, est-ce la fin des buffer overflows ? Oui et non. Ces attaques restent bien possibles, bien que plus difficiles et coûteuses à mettre en place, mais ne sont en réalité qu'un outil à la ceinture d'un attaquant. De nombreuses autres vulnérabilités existent, au même niveau comme les integer overflows ou les use-after-free, ou à des niveaux supérieurs comme des injections SQL ou des failles XSS. Néanmoins, l'apparition de nouveaux langages compilés et memory-safe comme Rust ou Go permettent tout de même de drastiquement limiter ces vecteurs d'attaques liés à la mémoire.