Les pièges du C

par Nicolas Joseph

Date de publication : 18 Avril 2006

Dernière mise à jour :

I - Introduction

Le C est un langage proche de la machine (manipulation d'adresses mémoire) et le compilateur laisse une grande liberté au développeur. De ce fait, écrire un programme en C demande beaucoup de rigueur. Cependant écrire un code propre et sans comportement indéfini n'est pas suffisant. En effet, l'intérêt pour l'informatique serait bien moindre sans l'interactivité homme-machine. Hélas s'il y a un point sur lequel on ne peut faire aucune supposition c'est bien la réponse que va fournir un utilisateur à notre programme. Dans un monde idéal où tous les utilisateurs seraient voués de bonnes intentions, une mauvaise réponse entraînerait dans le pire des cas, un arrêt brutal du programme et une perte des données non enregistrées. Mais comme nous ne sommes pas dans un monde parfait, il existe des utilisateurs mal intentionnés et les conséquences peuvent être bien plus graves qu'un simple plantage. Et bien souvent le but recherché par les pirates est d'obtenir les pleins droits sur une machine (un shell avec les droits root sur un système de type Linux par exemple) ou encore lancer l'exécution d'un virus. Maintenant que j'ai essayé de vous convaincre de la dangerosité d'un programme possédant des failles, je vais vous décrire les types de failles les plus connus et surtout la manière de les éviter. Il est peut être un peut tard mais je m'excuse auprès des personnes qui lisent cet article dans le but de pirater un ordinateur, en effet il n'y a aucune explication à ce sujet et histoire de vous encourager à allez voir de quoi il s'agit, sachez que pour espérer mettre ceci en pratique, il faut avoir quelques connaissances en C (audit de code), en assembleurs et en conceptions des systèmes d'exploitation : bon courage ;)

II - Dépassement de tampon

Aussi appelé buffer overflow, il s'agit sûrement du type de faille le plus connu et le plus simple à éviter. Lors d'un dépassement de tampon, l'utilisateur entre une chaîne de caractères qui va se retrouver dans un tableau de caractères de taille insuffisante ce qui va entraîner l'écriture de données en dehors de la zone mémoire allouée pour le tableau. Suivant le type de tableau (statique ou dynamique), on distingue deux types de buffer overflow :

- Stack overflow : dépassement de tampon qui va écraser la pile (tableau en mémoire statique)
- Heap overflow : dépassement de tampon qui va écraser le tas (tableau dynamique)

II-A - Sur la pile

Pour commencer voici un petit code qui illustre le problème :

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

static void cpy_arg (const char *);

int main (int argc, char **argv)
{
   if (argc == 2)
   {
      cpy_arg (argv[1]);
      printf ("argv[1] copier\n");
   }
   return 0;
}

static void cpy_arg (const char *s)
{
   if (s)
   {
      char text[20];

      strcpy (text, s);
   }
}
```

Les habitués auront tout de suite repéré que si l'utilisateur fourni comme argument à notre programme une chaîne de plus de 19 caractères, on obtient un dépassement de capacité lors de la copie dans *text*. Si le dépassement est suffisamment important, il est possible d'écraser des données système stockées dans la pile. Cette zone de mémoire doit son nom à la manière dont elle est utilisée : les données sont empilées lors de l'appel d'une fonction puis dépilée dans l'ordre inverse au retour dans la fonction appelante.

Les données stockées dans la pile sont :

- Les arguments de la fonction appelée
- Ses variables locales
- La valeurs des registres processeurs qui vont permettre, lors du retour de la fonction, de retrouver le contexte d'exécution de la fonction appelante

Ce sont ces registres qui vont nous intéresser puisque l'un d'entre eux est destiné à stocker l'adresse de la prochaine instruction à exécuter. Maintenant imaginez que dans notre fonction *cpy_arg*, on modifie cette valeur, au retour dans le *main*, le programme, à la place d'exécuter l'instruction *printf*, va chercher une instruction à l'adresse modifiée, qui peut pointer sur une zone mémoire où un pirate aura préalablement stocké un morceau de code de son cru (appelé *shellcode*). Ceci peut être facilement démontré :

```
$ gcc -Wall -W -O2 -ansi -pedantic bof.c
$ gdb a.out
...
(gdb) set args `perl -e "{print 'A'x512}"`
(gdb) run
Starting program: ./a.out `perl -e "{print 'A'x512}"`

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

Notre programme plante car il a essayé d'exécuter une instruction à l'adresse 0x414141 (0x41 est le code ASCII du caractère 'A'). On pourrait minimiser les risques d'une telle faille en se disant qu'il faut pouvoir entrer assez de données pour atteindre la pile. Hélas un débordement d'un byte peut permettre d'arriver au même résultat, on parle alors de *off-by-one overflow* (dans ce cas, plutôt que de modifier la valeur d'un registre placé dans la pile, on modifie la valeur d'une variable locale).

Pour conclure cette première partie, je vous propose un code que je trouve assez amusant et qui montre les véritables dangers d'une telle faille :

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

static void cpy_arg (const char *);
void code_pirate (void);

int main (int argc, char **argv)
{
   if (argc == 2)
   {
     printf ("code_pirate = %p\n", code_pirate);
     cpy_arg (argv[1]);
     printf ("argv[1] copier\n");
   }
   return 0;
}

static void cpy_arg (const char *s)
{
   if (s)
   {
      char text[20];

      strcpy (text, s);
   }
}

void code_pirate (void)
{
  printf ("Code pirate execute !\n");
}
```

On l'exécute :

```
$ a.out aaaabbbbccccddddeeeeffffgggghhhhiiiijjjjkkkk\x23\x13\x40
code_pirate = 00401323
Code pirate execute !
Segmentation fault
```

Et oh stupeur, la fonction *code_pirate* qui n'est jamais appelée dans notre code se retrouve exécutée ! Tout simplement parce qu'à la place d'écraser la valeur de la prochaine instruction à exécuter par une valeur quelconque (0x414141), on la remplace par l'adresse de la fonction *code_pirate* (little endian oblige, les octets sont entrés à l'envers).

Maintenant imaginez que l'adresse entrée n'est pas celle d'une fonction de notre programme mais celle d'une fonction présente dans une bibliothèque à liaison dynamique !

Dans ce cas d'école, le programme plante mais il est possible de le faire se terminer normalement : l'exécution du code est alors transparente pour l'utilisateur.

II-B - Sur le tas

Dans notre cas précédent, nous avons travaillé sur un tableau alloué en mémoire statique. Comme il existe des tableaux alloués en mémoire dynamique (fonction *malloc*, *calloc* et *realloc*) il existe aussi un dépassement de tampon qui va avec : le *heap overflow*. Cette fois le principe est un peu plus compliqué puisqu'il s'agit de modifier le contenu des structures de données utilisées par le système lors d'une allocation/désallocation de mémoire pour y stocker le code malicieux.

III - Format de chaîne de caractères

Qu'est ce qu'un format de chaîne de caractères ? Il s'agit du premier argument de la fonction *printf*, qui permet de spécifier le format de la chaîne de caractère à afficher (ce paramètre est présent chez toutes les fonctions de la famille de *printf*). Comme à chaque fois, le problème survient lorsque l'utilisateur a le contrôle sur le contenu de cette chaîne :

```
#include <stdio.h>

int main (int argc, char **argv)
{
   if (argc == 2)
   {
      char text[256];

      sprintf (text, "%s\n", argv[1]);
      printf (text);
   }
   return 0;
}
```

Le spécificateur de format *%x* permet d'afficher une valeur en hexadécimale :

```
$ ./a.out "%x"
8048524
```

En l'absence de paramètre, la fonction *printf* va afficher les valeurs présentes dans sa pile d'appel. Voici un exemple plus parlant :

```
$ ./a.out "AAAA|%x|%x|%x|%x|"
AAAAA|8048524|bffc1ac3|1|41414141|
```

Au bout d'un moment, on retrouve le début de notre chaîne! Tout cela n'aurait pas de conséquences désagréables s'il n'était pas possible d'écrire en mémoire. Hélas la fonction *printf* propose l'option *%n* qui permet de sauvegarder le nombre de caractère déjà écrit :

```
#include <stdio.h>

int main (void)
{
   int i = 0;

   printf ("Hello World!\n%n", &i);
   printf ("Cette phrase contient %d caracteres\n", i);
   return 0;
}
```

Ce qui donne :

```
$ ./a.out
Hello World!
Cette phrase contient 13 caracteres
```

On a bien écrit la valeur 13 à l'adresse *&i*. Eh bien essayons :

```
$ gdb a.out
...
(gdb) set args "AAAA|%x|%x|%x|%x|%n"
(gdb) run
Starting program: ./a.out "AAAA|%x|%x|%x|%x|%n"
```

```
Program received signal SIGSEGV, Segmentation fault.
0xb7e4b8a3 in vfprintf () from /lib/tls/libc.so.6
```

IV - Dépassement de capacité d'entier

Dernière vulnérabilité et sûrement la plus difficile à détecter. En effet cette faille utilise le fait que les nombres ne peuvent dépasser une certaine valeur qui dépend du nombre de bits utilisés pour leur représentation et de leur méthode de codage. Par exemple pour un **unisgned int** codé sur 32 bits en représentation binaire pur, la variable peut prendre une valeur allant de 0 à 4 294 967 295. La bibliothèque standard du C permet de retrouver ces valeurs grâce aux fichiers d'en-tête *limits.h* :

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

int main (void)
{
   unsigned int ul = UINT_MAX;
   printf ("ul=%u\n", ul);
   return 0;
}
```

Ce qui donne sur ma machine :

```
ul=4294967295
```

Mais que se passe-t-il si l'on dépasse cette valeur? Voici un exemple :

```
#include <stdio.h>

int main (void)
{
   unsigned int i = 1;

   while (i > 0)
   {
      i++;
   }
   printf("Hello world!\n");
   return 0;
}
```

D'un point de vu algorithmique, notre programme reste indéfiniment dans la boucle **while** mais si vous exécutez ce code suffisamment longtemps vous devriez voir s'afficher 'Hello world!'. Cela signifie que la valeur de *i* est devenue négative!

Pour vous montrer la difficulté de détecter ce genre de faille, voici un code vulnérable :

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

int main (int argc, char **argv)
{
    int ret = EXIT_FAILURE;

    if (argc == 2)
    {
        char buf[255];
        unsigned char size = strlen (argv[1]);

        if (size < 254)
        {
            printf ("size = %d\n", size);
            strcpy (buf, argv[1]);
            ret = EXIT_SUCCESS;
        }
    }
```

```
    return ret;
}
```

Après l'avoir compilé avec le mode parano de gcc :

```
$ ./a.out `perl -e "{print 'A'x500}"`
size = 244
Erreur de segmentation
```

Eh oui, la valeur 500 stockée dans un **unsigned char** (codé sur 2 octets sur ma machine), se transforme en 244 et nous permet de copier 500 caractères dans une tableau prévu pour en contenir que 255 et l'on se retrouve dans le cas classique d'un buffer overflow!

V - Conclusion

Comme vous avez pu le constater, bien configurer son compilateur ne suffit pas pour obtenir du code sûr. Il faut aussi privilégier les fonctions qui tiennent compte de la taille de leurs arguments, telles que fgets, strncpy ou encore strncat pour éviter tout débordement. Cependant pour ces deux dernières, il ne s'agit pas d'une solution idéale puisque que si la taille de la chaîne dépasse celle spécifiée en paramètre, le 0 de fin de chaîne n'est pas ajouté (il suffit de le faire manuellement, encore faut-il y penser).

Il semble tout de même utopique de n'exécuter que des programmes dont le code est sans faille. Par conséquent, plutôt que passer en revue tous les codes (à supposer que le code soit disponible) pour le sécuriser afin d'éviter l'exécution de code malveillant, il faut penser à une autre solution : sécuriser l'environnement d'exécution des programmes.

Par exemple sous Linux, il existe un ensemble de patch pour le noyau : GrSecurity qui contient, entre autres, le patch PaX (Protection against eXecution) qui limite les conséquences des failles vu dans cette article. Bien sûr ce genre de protection n'est pas une excuse pour laisser des failles dans vos programmes puisqu'elle n'est pas invulnérable mais permet de rendre la tâche du pirate (beaucoup) plus compliquée.

Comme je l'ai précisé en introduction, je n'ai abordé que les failles les plus connues (de moi surtout) donc si vous en connaissez d'autres, je les intégrerai à cet article avec plaisir ;) Pour me contacter, vous trouverez un lien pour m'envoyer un MP sur la page de l'équipe C & C++.

VI - Remerciements

Merci à neguib pour la relecture attentive de cet article.