DEBORDEMENT D'ENTIER

(INTEGER OVERFLOW)

Cindy MALATERRE Déborah JOURDES

3 décembre 2007

Table des matières

Chapitre 1

Introduction

1.1 Qu'est-ce qu'un entier ?

Un entier est un nombre, sans partie décimale. Dans le contexte informatique, ils sont représentés par des variables de la même taille qu'un pointeur sur le système, i.e selon l'architecture, 32 ou 64 bits.
Il existe deux types numériques pour les entiers (*long*, *int* et *short*) :

- Le type *signed*, pour les entiers signés, qui peuvent prendre des valeurs positives ou négatives.
  Le signe est déterminé par le bit de poids le plus fort : si celui-ci vaut 0 (resp 1), l'entier est interprété comme étant positif (resp négatif).
  La valeur maximale qui peut être représentée par un *int* (resp *short*), signé par défaut, est (en architecture 32 bits) VMS $:=2^{31}-1$ (resp $2^{15}-1$). La valeur minimale est -2^{31} (resp -2^{15}).
- Le type *unsigned*, pour les entiers non signés, qui ne prennent que des valeurs positives.
  La valeur maximale d'un *unsigned int* (resp *unsigned short*) est donc VMU $:=2^{32}-1$ (resp $2^{16}-1$).

1.2 Qu'est-ce qu'un débordement d'entier ?

Un débordement d'entier a lieu lorsque l'on tente d'affecter à une variable entière, une valeur supérieure à celle qu'elle peut recevoir.
Le standard ISO C99 dit qu'un débordement d'entier cause un "comportement indéfini", ce qui signifie que les compilateurs peuvent faire ce qu'ils veulent. La plupart des compilateurs semblent ignorer le débordement, aboutissant à un résultat stocké inattendu ou erronné.

1.3 Quels sont les risques ?

Le danger d'un débordement d'entier vient du fait qu'il ne peut pas être détecté après qu'il se soit produit.
Le risque est donc qu'un résultat erroné peut être utilisé pour gérer la taille d'un tampon, une limite (de boucle ou d'index de tableau), ou pour provoquer un bug de signe.
Un entrée de type *integer* peut donc être utilisée par un attaquant pour déclencher un débordement du tas avec ses propres données, ou pour lire de l'information qu'il ne fournit pas (en utilisant les données utilisateur).

Chapitre 2

Débordement d'entier

2.1 Principe

Lorsqu'un débordement d'entier se produit, ISO C99 dit que cela peut provoquer un comportement inattendu. Cependant, ISO C99 ajoute que pour les entiers non signés, il ne peut y avoir de débordement, car ils sont calculés modulo VMU+1. C'est ce que l'on appelle le *"wrap around"*.

```
   ex1 : wrap_around.c
/*************************************************************
Les limites de la représentation des entiers.
Ce programme permet de voir les différences entre les entiers
signés et les non signés grâce aux options -s et -u.
*************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[]) {
  unsigned short non_signe;
  short signe;
  if(argc != 3){
    usage(argv[0]);
    return 1;
    }
  if(strcmp(argv[1],"-u", 2) == 0){
    printf("\n taille : %d bits \n arg : %s \n",sizeof(non_signe)*8,argv[2]);
    sscanf(argv[2],"%hu",&non_signe);
```

```
    printf("\n valeur non_signe : %hu \n", non_signe);
    }
  else if(strncmp(argv[1], "-s", 2) == 0){
    printf("\n taille : %d bits \n arg : %s \n",sizeof(signe)*8,argv[2]);
    sscanf(argv[2],"%hd",&signe);
    printf("\n valeur signe : %hd \n", signe);
    }
  return 0;
}
```

Résultats des tests, pour des valeurs "limites" :

```
cindy@nana :~/Master2/SL$ ./wrap_around -s 32767
taille : 16 bits
arg : 32767
valeur signe : 32767

cindy@nana :~/Master2/SL$ ./wrap_around -s 32768
taille : 16 bits
arg : 32768
valeur signe : -32767

cindy@nana :~/Master2/SL$ ./wrap_around -u 65535
taille : 16 bits
arg : 65535
valeur non_signe : 65535

cindy@nana :~/Master2/SL$ ./wrap_around -u 65536
taille : 16 bits
arg : 65536
valeur non_signe : 0

cindy@nana :~/Master2/SL$ ./wrap_around -u 65537
taille : 16 bits
arg : 65537
valeur non_signe : 1
```

Les débordements d'entiers sont souvent le résultat d'opérations arithmétiques :

```
   ex2 : arithmétique.c
/*************************************************************
Les opérations arithmétiques de base sur les entiers signés et non
signés.
*************************************************************/
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  short s1,s2;
  unsigned short us1, us2;
  s1=10000;
  s2=7;
  us1=10000;
  us2=10;
  printf("\n débordement suite à une multiplication \n");
  printf("\n s1= %hd \n s2= %hd \n s1*s2=%hd",s1,s2,s1*s2);
  printf("\n us1= %hu \n us2= %hu \n us1*us2=%hu",us1,us2,us1*us2);
  s1=17000;
  s2=18000;
  us1=33000;
  us2=34000;
  printf("\n débordement suite à une addition \n");
  printf("\n s1= %hd \n s2= %hd \n s1+s2=%hd",s1,s2,s1+s2);
  printf("\n us1= %hu \n us2= %hu \n us1+us2=%hu",us1,us2,us1+us2);
  return 0;
}
```

Résultats des tests :

```
cindy@nana :~/Master2/SL$ ./arithmetique
debordement suite a une multiplication

s1 = 10000
s2 = 7
s1*s2 = 4464
us1 = 10000
```

```
us2 = 10
us1*us2 = 34464
debordement suite a une addition

s1 = 17000
s2 = 18000
s1+s2 = -30536
us1 = 33000
us2 = 34000
us1+us2 = 1464
```

Un débordement d'entier est la conséquence d'une tentative de stockage d'une valeur dans une variable qui est trop petite pour la contenir.
Pour illustrer ce phénomène, l'exemple le plus simple est le suivant :
Essayons d'affecter la valeur d'une variable de type *int* à une variable de type *short*.

```
   ex3 : perte_precision.c
/*****************************************************
          Exemple de promotion.
*****************************************************/
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  int l;
  short s;
  l=0xc1dbeaef;
  s=l;
  printf("\n l=0x%x ou %d (%d bits) \n",l,l,sizeof(l)*8);
  printf("\n s=0x%x ou %d (%d bits) \n",s,s,sizeof(s)*8);
  return 0;
}
```

Résultats des tests :

```
cindy@nana :~/Master2/SL$ ./perte_precicion
l = 0xc1dbeaef ou -1042552081 (32 bits)
s = 0xffffeaef ou -5393 (16 bits)
```

On doit maintenant introduire une nouvelle notion, celle de la promotion : Si les opérandes d'un calcul sont de tailles différentes, la taille de l'opérande le plus petit est étendue à celle du plus grand, pour que le calcul puisse avoir lieu.
Ensuite, le contenu de la variable "promue" est ajustée (ici à 16 bits) dans le but d'être stocké. Il est donc parfois nécessaire de tronquer le résultat, s'il est supérieur à la valeur maximale gérable. C'est ce qui le peut fausser.

2.2 Exploitation

Contrairement à la plupart des classes de bugs, les débordements d'entiers n'autorisent pas un écrasement direct de la mémoire, mais sont plus subtils. Il y a donc un grand nombre de situations dans lesquelles ils peuvent être exploités.
Nous allons en présenter deux exemples.

2.2.1 Débordement de tampon

Le plus souvent, les débordements arithmétiques sont exploités quand un calcul est effectué pour déterminer la taille d'allocation d'un tampon.
Par exemple, lorsqu'un programme doit allouer de la mémoire pour un tableau (avec malloc), il peut le faire par un calcul en multipliant le nombre d'éléments par la taille d'un objet.
Comme on l'a vu précédemment, si l'on peut contrôler un de ces deux opérandes, le calcul peut alors produire un résultat erroné et ainsi la taille du tampon est incorrecte.

```
   ex4 : fonction copie_boucle
/*************************************************************
Cette fonction copie lgr éléments de tab dans copie_tab.
*************************************************************/
int copie_boucle(int *tab, int lgr) {
  int *copie_tab, i ;
  copie_tab = malloc(lgr*sizeof(int)) ;      /*[1]*/
  if(copie_tab == NULL) {
    return -1 ;
    }
```

```
  for(i=0 ;i≤lgr ;i++) {      /*[2]*/
     copie_tab[i]=tab[i] ;
     }
  return copie_tab ;
}
```

Cette fonction à priori inoffensive peut conduire à la fermeture du programme.
En effet, il est possible de forcer le tampon à avoir la taille que l'on souhaite en soumettant une valeur de `lgr` assez grande à la multiplication `[1]`, alors dans la boucle `[2]`, le programme écrit à la fin du tampon `copie_tab`, il y a débordement du tas.

2.2.2 Bugs de signes

Qu'est ce qu'un bug de signe ?

Dans la mémoire d'un ordinateur, il n'y a pas de distinction entre la façon dont les variables signées et non signées sont stockées, ce qui peut mener à des confusions d'interprétation. Lorsqu'une variable non signée (resp signée) est interprétée comme signée (resp non signée), on parle de bug de signe.

```
   ex5 : exemple classique de bug de signe
int copie_memcpy(char *tmp, int lgr) {
  char copie_tmp[800] ;
  if(lgr≥sizeof(copie_tmp)) {      /*[1]*/
     return -1 ;
     }
  return memcpy(copie_tmp, tmp, lgr) ;      /*[2]*/
}
```

Ici, le problème est que `memcpy` prend comme paramètre un entier non signé mais dans la vérification `[1]`, `lgr` est de type *signed int* et `sizeof` retourne un *unsigned int*. Donc, si on prend une valeur négative pour `lgr`, on va passer la vérification `[1]`, or dans l'appel à `memcpy`, `lgr` sera interprétée comme une très grande valeur (non signée) provoquant alors l'écrasement de la mémoire à la fin du tampon `copie_tmp`.

Une confusion entre signés et non signés peut conduire à un autre problème :

```
   ex6 : confusion de signe
/*********************************************************************
Cette fonction insère un élément val, à la position pos de tab.
*********************************************************************/
int tab[800] ;
int inserer(int val, int pos) {
  if(pos> (sizeof(tab) / sizeof(int)) ) {
     return -1 ;
     }
  tab[pos]=val ;
  return 0 ;
}
```

La ligne `tab[pos]=val` étant équivalente à `*(tab + (pos*sizeof(int)))=val`, il paraît évident qu'une valeur négative n'est pas attendue pour `pos`, sachant que, en temps normal `tab + pos` doit être supérieur à `tab`.
Donc passer une valeur négative pour `pos` entraînera une situation inattendue que le programme ne pourra pas gérer.

Bug de signe causé par un débordement d'entier

Les bugs de signe peuvent être causés par un débordement d'entier , cependant cette classe de bug peut être assez problématique à exploiter car, les entiers signés interprétés en non signés tendent à être énormes, et on est donc souvent confronté à des *segfault*.
L'exemple suivant montre ce qu'il peut parfois se produire dans les démons réseaux, lorsqu'une indication de taille est envoyée dans un paquet par un utilisateur à qui l'on ne fait pas forcément confiance.

```
   ex7 : bug de signe dû à un débordement d'entier
int get_two_vars(int sock, char *out, int len) {
  char buf1[512], buf2[512] ;
  unsigned int size1,size2 ;
  int size ;
```

```
  if(recv(sock,buf1,sizeof(buf1),0)<0) {
    return -1;
    }
  if(recv(sock,buf2,sizeof(buf2),0)<0) {
    return -1;
    }
  memcpy(&size1, buf1, sizeof(int));    /*le packet commence par
  memcpy(&size2, buf2, sizeof(int));    une information de taille*/
  size = size1 + size2;       /*[1]*/
  if(size>len) {      /*[2]*/
    return -1;
    }
  memcpy(out, buf1, size1);
  memcpy(out + size1, buf2, size2);
  return size;
}
```

Ici, l'addition `[1]` est censée empêcher que les données excèdent les limites du tampon de sortie. Néammoins, en choisissant judicieusement les valeurs de `size1` et `size2`, on peut faire en sorte que la variable `size` ait une valeur négative.
On pourrait prendre par exemple `size1` = `size 2` = `0x7fffffff` ($=2^{31}-1$). Dans ce cas, on passe la vérification `[2]` car `0x7fffffff` + `0x7fffffff` = `0xfffffffe` (=-2), et la taille du tampon qui peut être écrit sera bien plus importante que prévue ($2^{32}-2$ au lieu de `len`). Le second appel à `memcpy` ayant pour destination `out + size1` pourra alors permettre (selon le choix de `size1`) d'atteindre n'importe quel endroit de la mémoire.

Chapitre 3

Solutions

La cause première des débordements d'entiers est l'insuffisance des vérifications pour les entrées de type *integer*.
Plusieurs situations sont particulièrement exploitables et donc à surveiller. Il faut alors essayer de :

1. veiller à ce que de l'allocation de mémoire ne se fasse pas directement sur un paramètre d'entrée donné par l'utilisateur.
2. maîtriser les compteurs d'itérations.
   Il est fréquent de rencontrer dans un programme, des boucles explicites (resp implicites) de la forme :
   `if(cptr<limite){instructions ; cptr++ }`
   (resp `if(cptr<limite){ fonction(cptr)}`, par exemple avec `memcpy`).
   Nous avons vu qu'il était possible que `cptr`, s'il est un entier signé, passe le test (un entier négatif par exemple), pour être ensuite interprété comme non signé dans la boucle ou la fonction.
   Un contrôle supplémentaire pourrait être mis en place, il faudrait fixer un `seuil` maximal d'itérations, indépendant de la boucle.
   Ainsi, si un attaquant veut déborder un tampon en utilisant une *faille integer*, alors l'entier qu'il devra envoyer sera un énorme nombre. Par exemple :

   - dans le cas d'une allocation de tampon pour les données utilisateur : `tmp = malloc(lgr + sizeof(en_tête))`, `lgr` doit être de l'ordre de 2^{32} etant donné que la taille d'une structure dépasse rarement quelques octets.
   - dans le cas d'une allocation d'un tableau :
     `tmp = malloc(lgr * sizeof(objet))`, `lgr` est plus petit que dans

l'exemple précédent mais reste relativement grand.

Par exemple, si `sizeof(objet)` = 4, `lgr` doit être de l'ordre de 2^{30}.

– dans le cas où l'on souhaite tomber dans l'ordre négatif :

On devra envoyer un nombre supérieur à 2^{31}.

On voit alors que si le nombre d'itérations est inférieur à `seuil` :=2^{30}, on devrait éviter la majorité des exploits. On peut choisir une valeur pour `seuil` plus petite pour une meilleure protection, mais qui doit rester suffisamment grande pour que les boucles légitimes puissent avoir lieu.

Il existe un patch gcc, implémentant cette solution : blip (Big Loop Integer Protection).

Le compilateur gcc étant au courant de toutes les boucles dans l'application, il lui sera possible d'ajouter les tests de sécurité appropriés avant toute boucle (notamment `limite` $\leq$ `seuil`). Ceci sécurisera l'application sans aucune connaissance spécifique sur la faille.

Conclusion

Un débordement d'entier n'est pas un phénomène dangereux en lui-même. Cependant, il représente un risque important s'il est bien exploité. La présence de *failles integer* dans les démons réseaux et les kernel system a fait que les recherches se sont multipliées dans ce domaine. Les débordements d'entiers étant indétectables après s'être produits, les solutions qui sont apparues permettent seulement d'éviter au maximum qu'ils aient lieu.

Bibliographie

[1] *Basic integer overflow* Blexim, Phrack 60, 2002.

[2] *Big loop integer protection* Oded Horovitz, Phrack 60, 2002.

[3] *Integer Overflow* Wikipedia, http ://en.wikipedia.org/wiki/Integer_overflow.