Systèmes d'Exploitation

Débordement de buffer

Didier Verna

didier@lrde.epita.fr
http://www.lrde.epita.fr/~didier

Table des matières

Introduction

Buffer Overflow

Mémoire

Funcall

Overflow

Shell

Exploitation

Conclusion

1 Introduction

2 Buffer Overflow
- Structure de le mémoire
- Appels de fonction
- Buffer overflow

3 Lancement d'un shell

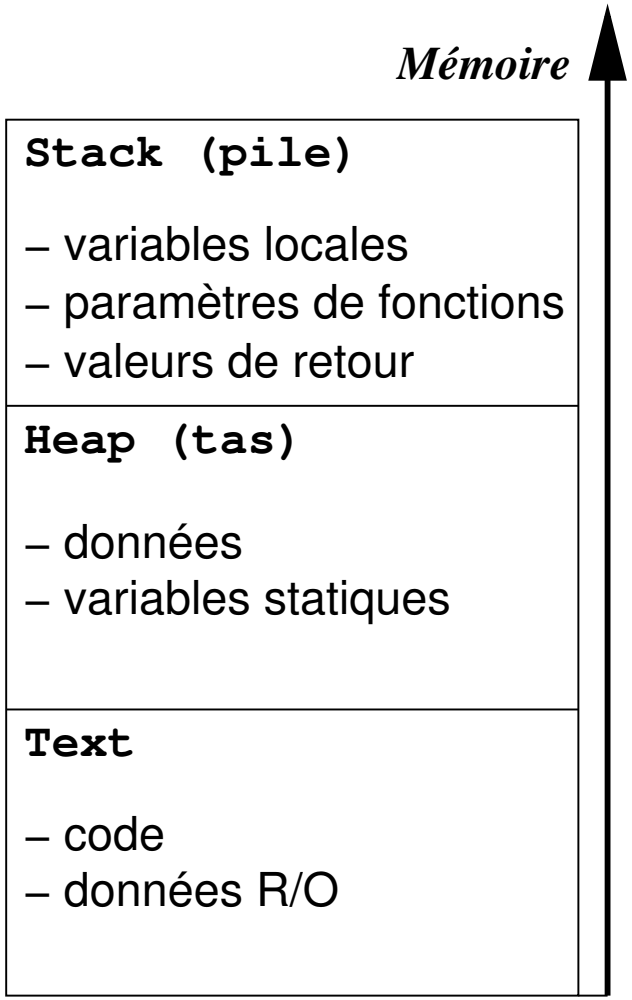
4 Exploitation

5 Conclusion

Vive le C !

- **Déficience du langage C**
  - **Débordement de buffer** (« buffer overflow ») : il est possible d'écrire dans un tableau déclaré `auto` plus loin que sa taille réelle.
  - **Corruption de pile** (« stack smashing ») : un débordement de buffer peut corrompre la pile d'exécution et changer l'adresse de retour d'une fonction.
- Bugs parmis les plus difficiles à détecter / corriger
- **Exploitation**
  - Profiter d'une corruption de pile pour faire exécuter du code illicite par le programme défaillant (ex. un shell).
  - Failles de sécurité (malheureusement) très répandues (`mount`, `sendmail`, `Xt...`).
  - Seule protection possible : programmer correctement.

Structure de la mémoire processus



- Toute tentative d'écriture dans le segment `Text` déclenche un SIGSEGV.
- Le tas est retaillable (`brk(2)`). L'ajout de mémoire se fait entre le tas et la pile.
- La pile est implémentée en LIFO.

Structure de pile

- **Description**
  - Zone de mémoire contigüe. Base fixe, registre `SP` pointant vers le sommet.
  - Taille réajustée dynamiquement par le noyau.
  - Instructions `push` et `pop` fournies par le jeu d'instructions machine.
- **Stack Frames**
  - Unités logiques empilées à chaque appel de fonction, dépilées à chaque sortie de fonction.
  - Contiennent les paramètres d'appels, les variables locales, et de quoi retrouver la frame précédente.
- **Frame pointer (`FP`)** (Local Base Pointer)
  - Pointeur sur chaque frame en plus du pointeur de pile.
  - Fourni un référencement constant au sein de chaque frame (insensible aux modifications de la pile).

Variantes d'implémentation

- La pile peut être orienté dans le sens ascendant ou descendant (Intel, Motorola, Sparc, MIPS) vis-à-vis de la mémoire.
- Le pointeur de pile peut pointer sur la dernière adresse utilisée, ou sur la première adresse libre.
- Le frame pointer est stocké dans le registre `BP` sur Intel, n'importe où sauf `A7` sur Motorola.

Appel de fonction

« Prologue » ⟶ fonction ⟶ « Épilogue »

Systèmes d'Exploitation

Didier Verna
EPITA

Introduction

Buffer Overflow

Mémoire

Funcall

Overflow

Shell

Exploitation

Conclusion

- **Prologue**
  - Sauvegarde de `FP` (pour restauration future dans l'épilogue).
  - Copie de `SP` dans `FP` (la nouvelle frame est crée au sommet de la pile).
  - Déplacement de `SP` (pour l'allocation effective de l'espace de la nouvelle frame).
- **Instructions dédiées**
  - `enter` et `leave` sur Intel.
  - `link` et `unlink` sur Motorola.

Exemple

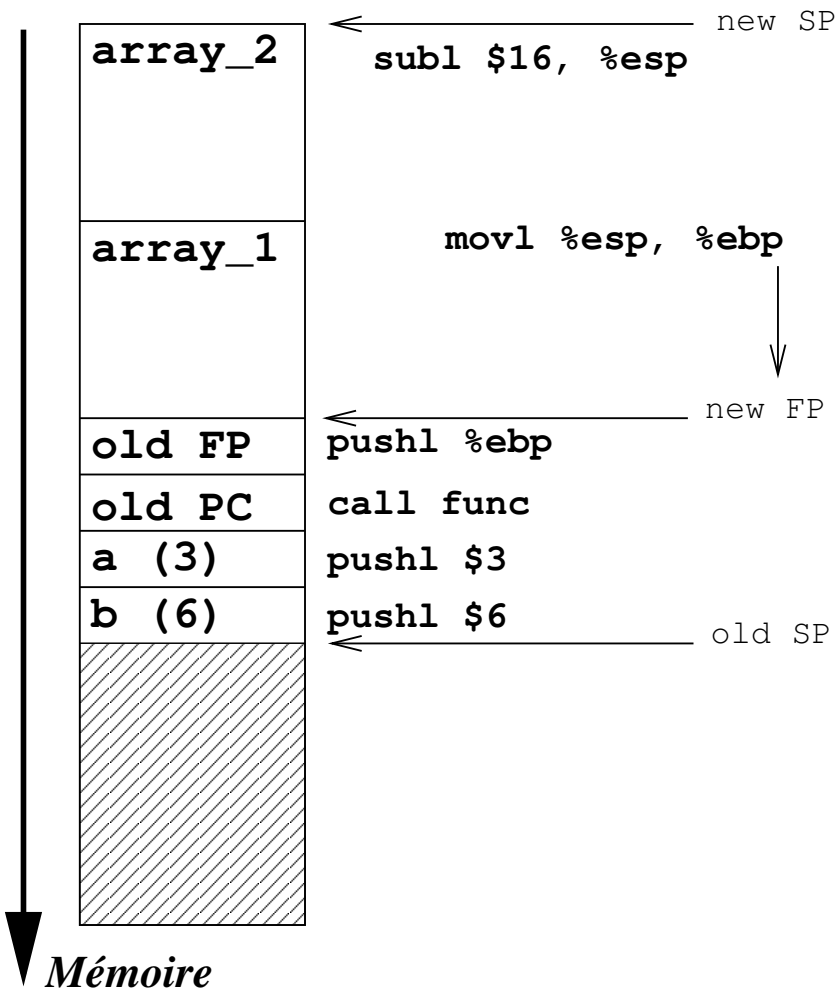
Source :

```
static void func (int a, int b)
{
  char array_1[8];
  char array_2[8];
}

int main (int argc, char *argv[])
{
  func (3, 6);

  return 0;
}
```

gcc -S :

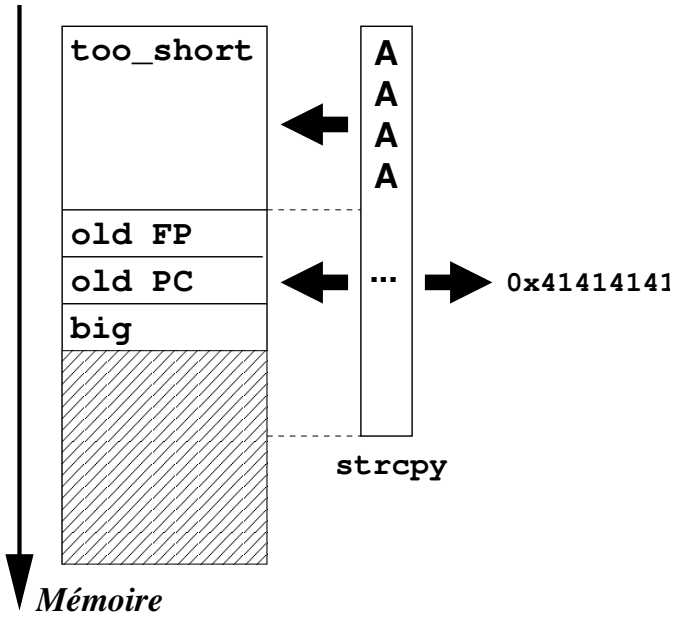
```
func:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        leave
        ret
main:   ...
        pushl   $6
        pushl   $3
        call    func
        ...
        leave
        ret
```

Évolution de la pile



Buffer overflow

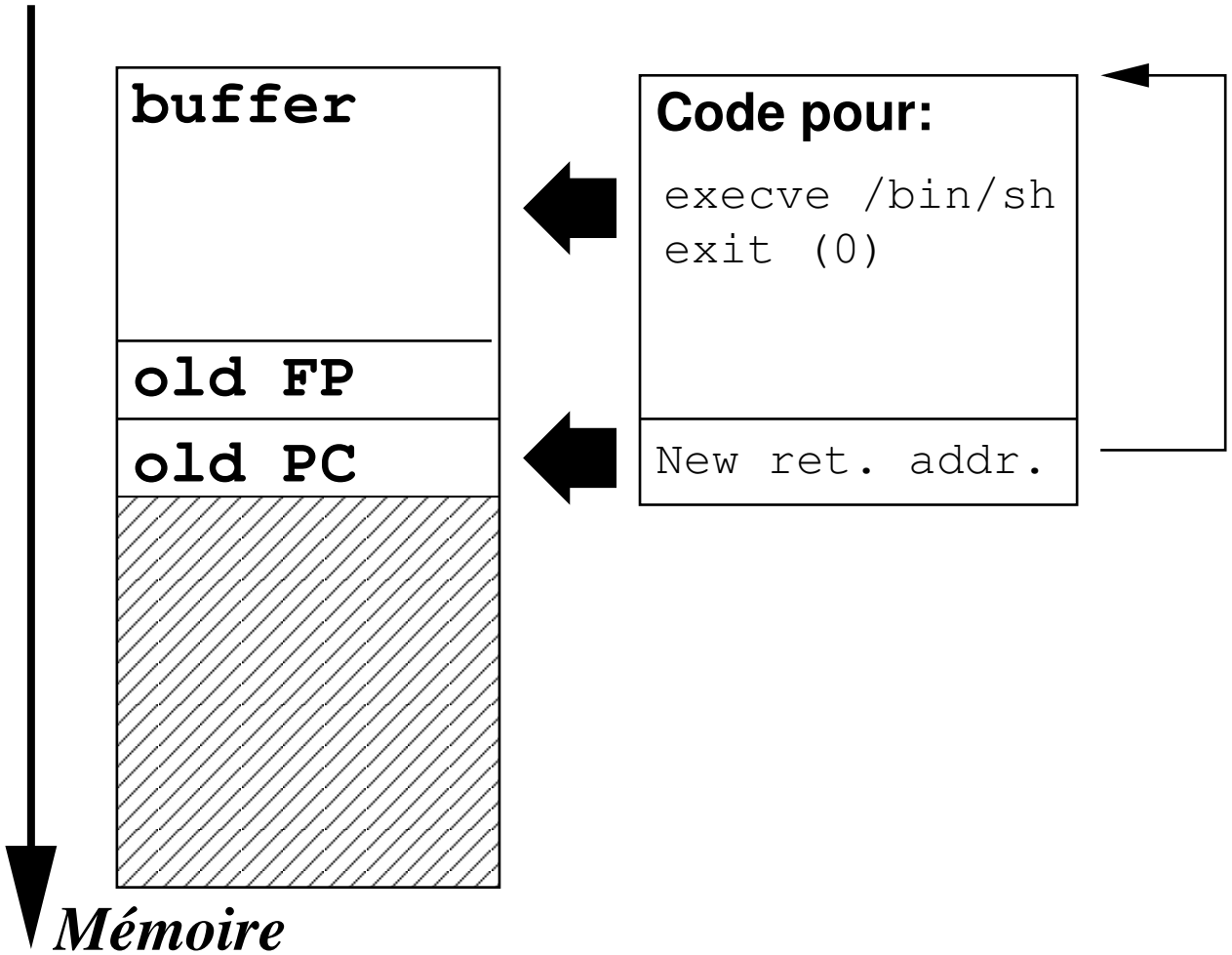

```
static void func (char *big)
{
  char too_short[8];

  strcpy (too_short, big);
}

int main (int argc, char *argv[])
{
  char big[16]
    = "AAAAAAAAAAAAAAAA";

  func (big);
  return 0;
}
```



Détournement de code

- **Effet de bord**
  - Un débordement de buffer écrase la pile
  - La valeur de retour (`old PC`) peut donc être écrasée
  - On peut donc diriger le flot d'exécution sur du code arbitraire
- **Exploitation**
  - Charger le code requis dans le buffer concerné (ex. shell)
  - S'arranger pour que la valeur de retour pointe sur ce code

Préparation



Récupération de l'assembleur

Systèmes d'Exploitation

Didier Verna
EPITA

Introduction

Buffer Overflow

Mémoire

Funcall

Overflow

Shell

Exploitation

Conclusion

```c
#include <unistd.h>

int main(int argc, char *argv[])
{
  char *name[2];

  name[0] = "/bin/sh";
  name[1] = NULL;
  execve (name[0], name, NULL);
}
```

```c
#include <unistd.h>

int main (int argc, char *argv[])
{
  exit (0);
}
```

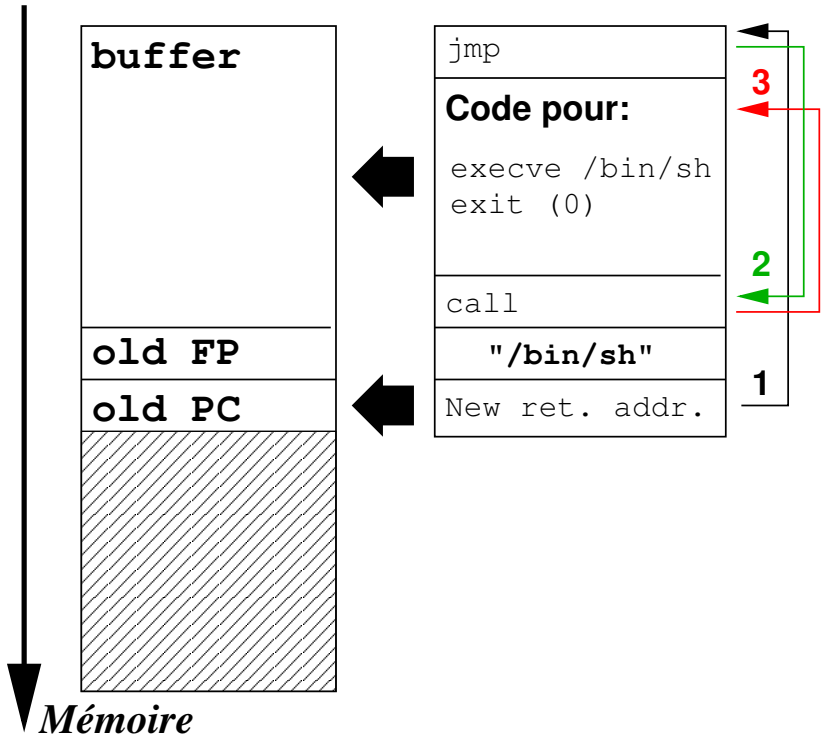
- Compilation statique pour obtenir le code des appels système
- Désassemblage avec un débugger

Adressage de la chaîne "/bin/sh"


```
main :
        push   %ebp
        mov    %esp,%ebp
        sub    $0x18,%esp
        ...
        movl   $0x8094da8,0xfffffff8(%ebp)
        ...
        call   0x804d0b0 <execve>
        leave
        ret
```

Problème : L'adresse de la chaîne "/bin/sh" n'est pas connue...

Solution : `jmp / call`



En plaçant un `call` juste au dessus de la chaîne de caractères, l'adresse de celle-ci sera poussée sur la pile comme valeur de retour.

Phase finale

Transformation du programme en chaîne de caractères

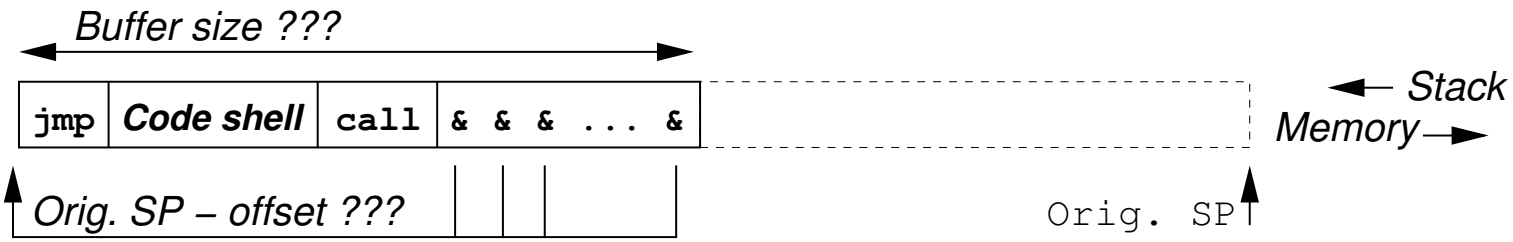
- Récupération de la sequence d'octets (débugger)
- Élimination des caractères nuls (`xorl %eax,%eax`)

```
char shellcode[] = "...";

int main (int argc, char *argv[])
{
  int *ret;

  ret = (int *) &ret + 2;
  (*ret) = (int) shellcode;
}
```

Jouons maintenant les gros vilains

- **Idée**
  - Trouver les buffer overflows (`strcpy...`)
  - Introduire le code (`argv`, `getenv...`)
- **Problèmes à résoudre**
  - Taille du buffer contenant le code
    $\Longrightarrow$ Trop petit : pas d'écrasement de la valeur de retour
    $\Longrightarrow$ Trop grand : risque de SIGSEGV
  - Localisation de la pile
    $\Longrightarrow$ Toutes les piles démarrent à la même adresse
  - Localisation du buffer sur la pile
    $\Longrightarrow$ Incantations mystiques

Solution



Pour obtenir l'adresse de la pile :

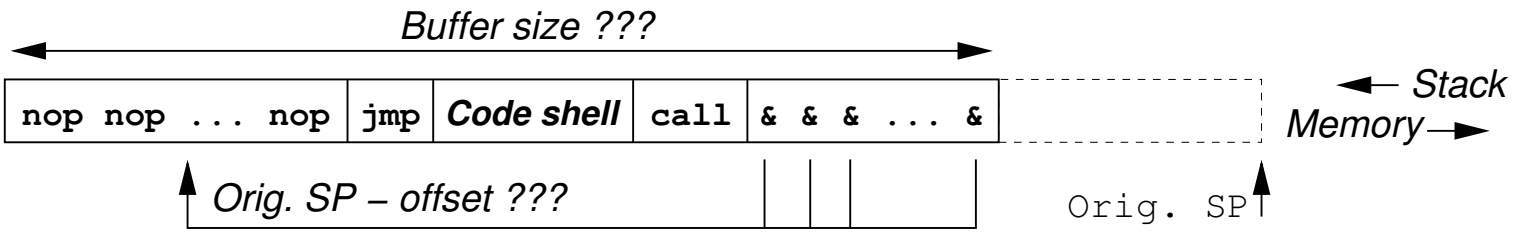
```
unsigned long get_sp (void)
{
  __asm__("movl %esp,%eax");
}
```

- Automatiser la fabriquation de la chaîne
- Paramétrer la taille du buffer et l'offset

Problème : l'offset doit tomber juste *à l'octet près*

Meilleure solution : `nop`

Permet d'être flou sur la véritable adresse du buffer



Bonnes conditions :

- Un buffer trop gros d'une centaine d'octets
- La moitié réservée pour les `nop`

Petits débordements

- **Quand le buffer est trop petit**
  - Pas assez de place pour tout le code
  - Pas assez de place pour les `nop`
- **Solution**

  L'état de la pile est connu au démarrage

strings	***argv pointers***	**NULL**	***env pointers***	**NULL**	**argc**	**argv**	**env**

  - Placer le code dans une variable d'environnement
  - Ecraser avec l'adresse de cette variable sur la pile

Programmez correctement !

Surveillez :

- `strcpy`, `strcat`...
- `sprintf`, `sscanf`, `vsprintf`...
- `getc`, `fgetc`, `getchar`... dans des boucles