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]="bbbbbbb";
}

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

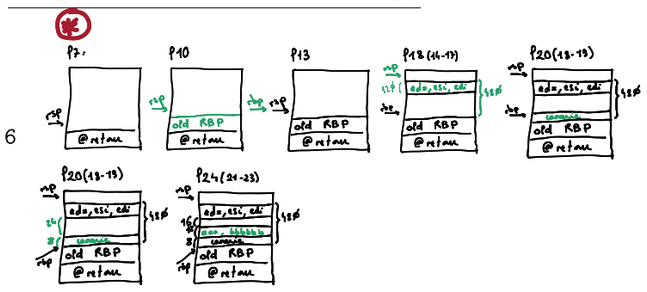
La traduction en assembleur du programme `tp1` peut être obtenue avec la commande `gcc` :

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

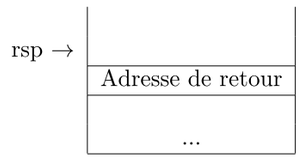
L'option `-g` a volontairement été omise pour ne pas surcharger l'affichage avec les options de debuggage. Voici un extrait du fichier `tp1.s` ainsi obtenu :

```
$ 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)
0x7fffffffe47c: 0x6262626200616161 0x595c480000626262
0x7fffffffe48c: 0xffffe4b0bc93deca 0x555551e000007fff
0x7fffffffe49c: 0xffffe5a000005555 0x0000000000007fff
0x7fffffffe4ac: 0x0000000000000000 0xf7de108300000000
0x7fffffffe4bc: 0xf7ffc62000007fff 0xffffe5a800007fff
(gdb) info frame             (<- information sur le contexte
                                d'execution courant)
Stack level 0, frame at 0x7fffffffe4a0:
 rip = 0x5555555551a2 in f (tp1.c:7); saved rip = 0x5555555551e0
 called by frame at 0x7fffffffe4c0
 source language c.
 Arglist at 0x7fffffffe458, args: a=1, b=2, c=3
 Locals at 0x7fffffffe458, Previous frame's sp is 0x7fffffffe4a0
 Saved registers:
  rbp at 0x7fffffffe490, rip at 0x7fffffffe498
(gdb) disas main
Dump of assembler code for function main:
   0x00005555555551b9 <+0>: endbr64
   0x00005555555551bd <+4>: push   %rbp
   0x00005555555551be <+5>: mov    %rsp,%rbp
   0x00005555555551c1 <+8>: sub    $0x10,%rsp
   0x00005555555551c5 <+12>: movl   $0x0,-0x4(%rbp)
   0x00005555555551cc <+19>: mov    $0x3,%edx
```

```
0x00005555555551d1 <+24>: mov    $0x2,%esi
0x00005555555551d6 <+29>: mov    $0x1,%edi
0x00005555555551db <+34>: callq  0x555555555169 <f>
0x00005555555551e0 <+39>: movl   $0x1,-0x4(%rbp)
0x00005555555551e7 <+46>: mov    -0x4(%rbp),%eax
0x00005555555551ea <+49>: mov    %eax,%esi
0x00005555555551ec <+51>: lea    0xe11(%rip),%rdi        # 0x555555556004
0x00005555555551f3 <+58>: mov    $0x0,%eax
0x00005555555551f8 <+63>: callq  0x555555555070 <printf@plt>
0x00005555555551fd <+68>: mov    $0x0,%eax
0x0000555555555202 <+73>: leaveq
0x0000555555555203 <+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.

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);
}

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 précisèment cette adresse, nous allons dans notre chaînes 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 :

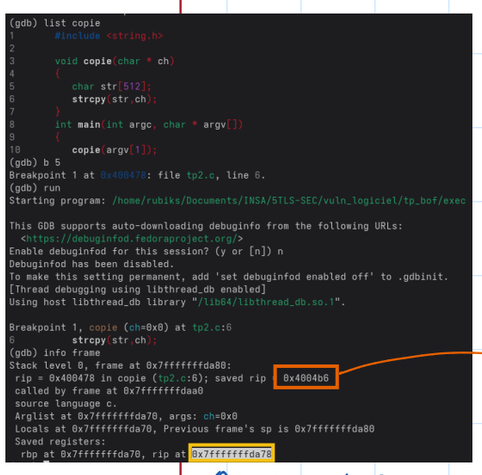
`NNNNNNNNNNNNSSSSSSSSSSSSSSSSSSAAAAAAAAAA`

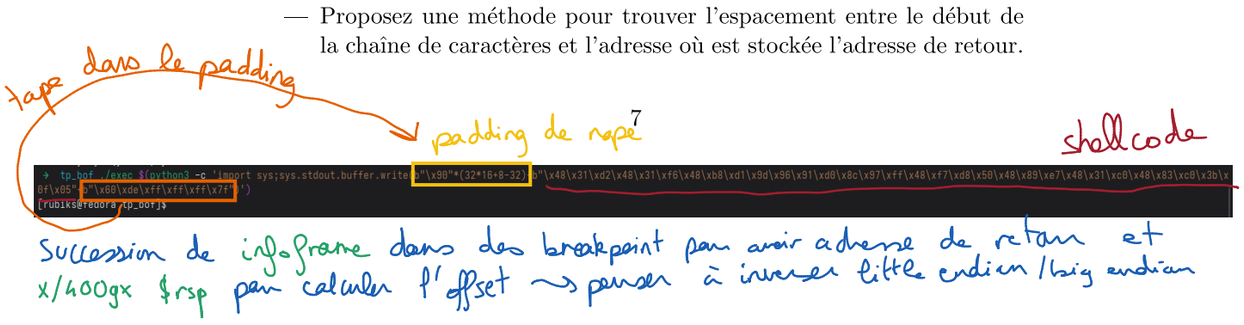
où :
- `N` est l'instruction NOP ;
- `SSSSSSSSSSSSSSS` est le shellcode ;
- `A` est l'adresse *supposée* de `str`.

Pour déterminer l'adresse `A`, comme on a desactivé 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.





- 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

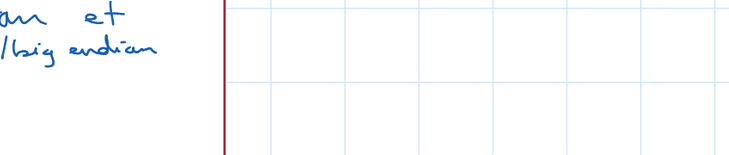
Pour réaliser ce TP, nous avons désactivé plusieurs mécanismes de protections. Nous allons les passer en revue et identifier leur utilité.

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.



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

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 ?

→ canarie ça détecte et plante

→ visiblement sur mon PC ça détecte pas ><

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.*

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/5TLS-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 0x7fffffffda80
 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.