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>
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.

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.
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 ?
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);

}
```

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(int d)
{
  printf("\nvaleur : %d\n",d);
}

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

```
  int 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`. Son exploitation va nous permettre de modifier la valeur de la variable `n`. Il faut pour cela déterminer son adresse et la recopier dans les 4 premiers octets de `buf`. Ainsi que nous l'avons montré en cours, ce sont ces 4 premiers octets qui vont être utilisés comme adresse mémoire pour satisfaire l'utilisation du format `%n`

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.
2. Exploitez ce programme à l'aide d'une chaîne de format, de façon à modifier la valeur de l'entier `n`.

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`.

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 4 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`.

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>

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

  int i=0;
  char out[256];

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

  memcpy(out, buf1, len1);
  memcpy(out + len1, buf2, len2);
  return 1;
}
```

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. Compilez votre programme avec les bonnes options (pour que `i` se trouve après `out`).

4. Lancez le programme de telle façon que vous choisissiez la valeur avec laquelle va être écrasé `i`.

3 Programmes SUID

Dans cette dernière partie, nous allons examiner deux petits exemples montrant comment un programme SUID peut être exploité si certaines données sont mal configurées, en particulier si ces programmes exécutent des commandes externes ou manipulent des fichiers.

3.1 Premier exemple

Dans ce premier exemple, nous allons considérer le cas d'un programme SUID-root qui fait une écriture dans un fichier.

```
#include <stdio.h>

int main()
{
  int euid=geteuid();
  int uid=getuid();
  FILE * fd;

  fd=fopen("/tmp/log","w");
  fprintf(fd,"%s","un message");
  fclose(fd);
}
```

1. Compilez et modifez les droits de ce programme de façon à ce qu'il appartienne à l'utilisateur `root` et à ce que le bit SUID soit positionné.
2. A l'aide d'un lien symbolique, montrez qu'un utilisateur quelconque (nous prendrons `debug` pour le TP) peut écrire dans n'importe quel fichier du système à l'aide de ce programme.

3.2 Second exemple

Dans ce second exemple, nous allons utiliser un programme SUID-root qui fait appel à une commande externe mais sans prendre suffisamment de précautions lors de cet appel.

Le code est le suivant :

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

int main(void)
{
  if (system ("mail $USER < fichier") != 0)
    perror ("system");
  return (0);
}
```

Ce code appelle la commande externe `mail` sans utiliser son chemin absolu. Et comme il est SUID-root, l'attaquant peut donc en profiter pour détourner son exécution et le faire exécuter du code sous l'identité de root. Pour cela, il suffit de créer un programme exécutable ou un script quelconque dont le nom est `mail` et de configurer la variable `PATH` de telle façon que ce programme soit appelé et non la commande du système.

1. Compilez le programme ci-dessus, changez son propriétaire à root et positionnez le bit SUID.
2. Créz un script shell dont le nom est `mail` et faites-le exécuter `/bin/sh`.
3. Lancez maintenant le programme ci-dessus en tant qu'utilisateur normal et vérifiez bien que vous obtenez un shell root.