Sommaire

Introduction

La format string est une vulnérabilité un peu particulière, les principes des buffer overflow ne peuvent pas être réutilisés et le principe même de l'écrasement de l'adresse de retour est particulier. Pour toutes ces raisons la format string est une vulnérabilité particulièrement intéressante à étudier !

1. Explications de l'origine du bug

Le bug se trouve dans la fonction printf, pour bien saisir ce qui se passe il faut déjà savoir comment s'utilise printf au niveau assembleur.

La fonction printf utilise en tant qu'argument des pointeurs. Le premier pointe vers une chaine de caractère et les suivants sont spécifiés dans la chaine de caractère.
Par exemple pour un printf("En hexa : %x", ValHexa) , le %x spécifie qu'il faudra affiche le prochain argument en tant nombre hexadécimal. Les %* sont nombreux, mais ici nous n'en auront besoin que de deux : %x et %n
C'est grâce au %n que l'on va pouvoir réécrire notre adresse de retour !
Il faut bien comprendre ce que fera %n pour la suite de l'article, tout repose sur cette instruction. %n compte le nombre de caractères ayant étés affichés à l'écran et stock le résultat dans Ecx, puis il récupère l'argument suivant dans Eax et écrit Ecx là où pointe Eax.
Si vous connaissez l'assembleur le code est le suivant :

```
MOV DWORD PTR DS:[EAX],ECX
```

Désolé pour ceux qui ne codent pas en assembleur mais il va falloir s'y mettre si vous voulez faire de la faille appli ;-) !

La faille provient d'une utilisation abusive de printf : printf(Variable)
La variable peut être par exemple votre pseudo. Donc si vous mettez comme pseudo "Heurs", la fonction sera printf("Heurs") . Là rien de grave nous sommes d'accord, mais nous mettons maintenant "Heurs%x", la fonction sera donc printf("Heurs%x") , mais nous n'avons spécifié aucun argument. Mais la fonction printf n'est pas censée le savoir, il affichera alors sont argument suivant, c'est à dire une variable empilée sur la stack.
Etant donné que les fonctions sont empilées sur la stack de façon croissante et que nous la lisons de façon décroissante, si on met plein de %x on affichera la stack en descendant, et logiquement à force de descendre on devrait retomber sur notre chaine de caractère (point très important). Je ne suis peut être pas très clair mais vous comprendrez surement mieux avec les exemples.
Rappelons nous du %n, ça veut dire que si on retombe sur notre chaine on pourrait très bien mettre un pointeur à la fin... et nous écririons donc le nombre de caractères affichés là où pointe ce pointeur. Je vous laisse déjà digérer ces concepts, nous allons voir lors de l'exploitation comment écrire une valeur voulue à la place de notre adresse de retour.

2. Exemples, analyse et exploitation

Codons déjà programme vulnérable :

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

int main(void) {
    int fd, i;
    char buffer[1024];

    fd = open("texte.txt", O_RDONLY);
    if (fd < 0) exit(0);
    read(fd, buffer, 1024);
    printf(buffer);
    close(fd);
    return 0;
}
```

Donc on ouvre texte.txt et on affiche ce qu'il contient (les 1024 premiers caractères seulement). Plaçons donc un simple "coucou !" dans texte.txt :

```
C:\FS_pratique>print.exe
coucou !
```

Maintenant on met quelques %x, "%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x" :

```
C:\FS_pratique>print.exe
23fb604004012be0000000107825782578257825782578257825782578257825782578257825782578
25
7825782578257825
```

Tiens tiens, on voit pas mal de \x25 \x78 qui se répètent. Si on converti en ASCII on s'aperçoit que 0x25 = % et que 0x78 = x. Donc nous sommes déjà retombé sur notre chaine de caractères !
Comme le %x fait 2 octets et que le pointeur d'arguments se déplace de 4 octets, on peut assez rapidement se mettre à l'extrémité de notre chaine. puis un petit %x%n suivit d'un pointeur et c'est presque fini ! Par exemple avec "%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%nbbbb" on a le droit à une belle exception. Je balance le code assembleur brute pour les habitués :

```
77C12AC4   8908              MOV DWORD PTR DS:[EAX],ECX

EAX 62626262
ECX 00000078
EDX 00033A9F
EBX 0000006E
ESP 0023F878
EBP 0023FAE4
ESI FFFFFFF8
EDI 0023FB90
EIP 77C12AC4 msvcrt.77C12AC4
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty -UNORM D0A8 01050104 00640079
```

```
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 037F  Prec NEAR,64  Mask    1 1 1 1 1 1
```

A ce stade nous pouvons constater que 0x62 = b et donc que si on replace bbbb par une adresse mémoire on écrira à cet endroit 0x78 (le nombre de caractères affichés). Si on veut afficher un certain nombre de caractère avec un %x on peu lui spécifier de la façon suivante : "%6x", ici nous afficherons 6 octets.
Ce qui serait cool ce serait de pouvoir réécrire l'adresse de retour avec une autre adresse mémoire (par exemple celle où se trouve un shellcode). Nous savons déjà comment pointer sur l'adresse (remplacer les bbbb par un pointeur sur l'adresse de retour) et maintenant nous savons comment y écrire la valeur que l'on souhaite ! Sisi, regardez par exemple pour écrire 0x0023FB94 il faudra faire un %2358164x (0x0023FB94 = 2358164). Bon il faudra un peu adapter le nombre suivant combien de caractères auront déjà étés affichés.
Notre exploit devra donc avoir la structure suivante :
`a%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%2358035x%n[pointeur sur l'adresse de retour][shellcode]`

Chez moi le nombre de caractères affiché me génère exactement l'adresse du début de mon shellcode. Ainsi j'ai modifié l'adresse de retour et saute sur mon shellde !

3. Sécurisation

Il faut tout simplement passer une chaine fixe en premier argument. Si on veut afficher une chaine de caractères un printf("%s", string) fera totalement l'affaire. En forçant le formatage on évite au hacker de se balader et réécrire notre stack.

Conclusion

Les format string ont de l'avenir par rapport aux buffer overflows, elles n'ont pas besoin d'écraser la pile ce qui a un avantage considérable quand on voit les moyens anti-buffer overflow qui sont déployés. Il en reste qu'elle ne sont pas des plus simples à exploiter et qu'il vous faudra faire beaucoup de tests afin de bien assimiler tout ce qui se passe.