# Creación de shellcodes BAJO LINUX

En la pasada entrega se aprendió a usar las syscalls de Linux en ensamblador. El objetivo es crear shellcodes que puedan ejecutarse más tarde cuando se explote alguna vulnerabilidad. Entiéndase por shellcode aquel código que ejecuta una shell (por ejemplo /bin/bash). Para fines del artículo se nombrará shellcode al código que ejecutará la syscall, aunque no necesariamente sea una shell.

Anteriormente se había creado un código en ensamblador que imprimía en pantalla la palabra "hola". Se muestra a continuación para refrescar la memoria del lector:

Archivo write.s :

```
.globl main
main:
mov $4,%eax          // Llamamos a la syscall write()
mov $1,%ebx          // File Descriptor es la salida estándar
push $0x616c6f68     // Empujamos "hola" en la pila
mov %esp,%ecx        // Movemos la dirección de la pila que apunta a "hola" en %ecx
mov $4,%edx          // 4 caracteres = 4 bytes
int $0x80            // Interruptor de ejecución

mov $1,%eax          //
mov $0,%ebx          // exit(0);
int $0x80            //
```

Se usa la etiqueta "main:" para que el compilador GCC la reconozca y pueda compilarla. Sin embargo, el binario que GCC produce está enlazado con muchas otras librerías, lo cual produce mucho código extra que no interesa. Así que de ahora en adelante sólo se usará el *ensamblador* para producir los binarios. Al usar el ensamblador puede usarse la etiqueta "_start:" en lugar de "main:" para que así pueda concentrarse solamente en la parte de interés. El código debe lucir como el siguiente:

Archivo write.s :

```
.globl _start
_start:
mov $4,%eax
mov $1,%ebx
push $0x616c6f68
mov %esp,%ecx
mov $4,%edx
int $0x80
mov $1,%eax
mov $0,%ebx
int $0x80
```

El objetivo es convertir este pedazo de código en ensamblador a instrucciones en lenguaje de máquina que el procesador pueda entender y ejecutar. Estas instrucciones están representadas por valores hexadecimales y son comúnmente llamados opcodes (Operation Codes). Para traducir el código en ensamblador a opcodes primero deben ensamblarse con el comando "as":

```
$ as write.s -o write.o
```

A continuación se usa el comando "objdump" para extraer los opcodes del objeto write.o:

```
$ objdump -d ./write.o
```

Este comando producirá una salida parecida a la siguiente:

```
Desensamblado de la sección .text:

00000000 <_start>:
   0:   b8 04 00 00 00          mov    $0x4,%eax
   5:   bb 01 00 00 00          mov    $0x1,%ebx
   a:   68 68 6f 6c 61          push   $0x616c6f68
   f:   89 e1                   mov    %esp,%ecx
  11:   ba 04 00 00 00          mov    $0x4,%edx
  16:   cd 80                   int    $0x80
  18:   b8 01 00 00 00          mov    $0x1,%eax
  1d:   bb 00 00 00 00          mov    $0x0,%ebx
  22:   cd 80                   int    $0x80
```

En la columna de la derecha se encuentra el código en ensamblador y en la columna izquierda se ve su equivalente en opcodes. Por lo tanto, la instrucción "**mov**

**$0x4,%eax**" es equivalente a los opcodes "**b8 04 00 00 00**". La instrucción "**mov $0x1,%ebx**" es equivalente a "**bb 01 00 00 00**" y así sucesivamente.

Con objeto de experimentación, se intentará ejecutar los opcodes desde un programa escrito en C, el cual seguramente en un futuro evolucionará en algún exploit. Para lograr esto deben concatenarse los opcodes en una cadena de caracteres anteponiéndole la cadena **'\x'** a cada opcode. La cadena de caracteres debe lucir de la siguiente forma:

```
Archivo write_shellcode.c:
char shellcode[] =  "\xb8\x04\x00\x00\x00"  //mov   $0x4,%eax
                    "\xbb\x01\x00\x00\x00"  //mov   $0x1,%ebx
                    "\x68\x68\x6f\x6c\x61"  //push  $0x616c6f68
                    "\x89\xe1"              //mov   %esp,%ecx
                    "\xba\x04\x00\x00\x00"  //mov   $0x4,%edx
                    "\xcd\x80"              //int   $0x80
                    "\xb8\x01\x00\x00\x00"  //mov   $0x1,%eax
                    "\xbb\x00\x00\x00\x00"  //mov   $0x0,%ebx
                    "\xcd\x80";             //int   $0x80
```

Para ejecutar el shellcode se llama a ejecución la cadena de caracteres dentro del código en C, así que el archivo write_shellcode.c quedaría de la siguiente forma.

Archivo write_shellcode.c:

```
char shellcode[] =  "\xb8\x04\x00\x00\x00"  //mov   $0x4,%eax
                    "\xbb\x01\x00\x00\x00"  //mov   $0x1,%ebx
                    "\x68\x68\x6f\x6c\x61"  //push  $0x616c6f68
                    "\x89\xe1"              //mov   %esp,%ecx
                    "\xba\x04\x00\x00\x00"  //mov   $0x4,%edx
                    "\xcd\x80"              //int   $0x80
                    "\xb8\x01\x00\x00\x00"  //mov   $0x1,%eax
                    "\xbb\x00\x00\x00\x00"  //mov   $0x0,%ebx
                    "\xcd\x80";             //int   $0x80

main() {
                    void (*fp) (void);

                    fp = (void *)shellcode;
                    fp();
}
```

Para comprobar que el shellcode se ejecute correctamente, se compila el código en C empleando GCC y se ejecuta:

```
$ gcc write_shellcode.c -o write_shellcode ; ./ write_shellcode
hola
```

Ahora se ha creado un shellcode funcional que imprime la cadena "hola". En la próxima entrega se descubrirá cómo optimizar este shellcode para hacerlo aún más pequeño y además se creará un shellcode que haga honor a su nombre, esto es, por supuesto, que ejecute una shell. ¡Hasta la próxima!