Creación de shellcodes bajo Linux: (By benn) benn@conthackto.com.mx

En la pasada entrega aprendimos a usar las syscalls de linux en ensamblador. Nuestro objetivo es crear shellcodes que podamos ejecutar mas tarde cuando explotemos alguna vulnerabilidad. Entiéndase por shellcode aquel código que nos ejecuta una shell (por ejemplo /bin/bash). Para fines del artículo llamaremos shellcode al código que ejecutará nuestra syscall, aunque no necesariamente sea una shell.

Anteriormente habíamos creado un código en ensamblador que imprimía en pantalla la palabra “hola”. A continuación lo mostramos 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         //
```

Usamos la etiqueta 'main:' para que el compilador GCC la reconociera y pudiera compilarlo. Sin embargo el binario que GCC produce esta enlazado con muchas otras librerías, lo cual produce mucho código extra que no nos interesa. Así que de ahora en adelante solo usaremos el *ensamblador* para producir nuestros binarios. Al usar el ensamblador podemos usar la etiqueta '_start:' en lugar de 'main:' para así poder concentrarnos solamente en la parte que nos interesa. 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
```

Nuestro 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 debemos ensamblarlo con el comando 'as':

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

A continuación usamos 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 encontramos el código en ensamblador y en la columna izquierda vemos su equivalente en opcodes. Por lo tanto la instrucción "**mov $0x4,%eax**" es equivalente a los opcodes "**b8 04 00 00 00**". La intrucción "**mov $0x1,%ebx**" es equivalente a "**bb 01 00 00 00**" y así sucesivamente.

Con objeto de experimentación intentaremos ejecutar los opcodes desde un programa escrito en C, el cual seguramente en un futuro evolucionara en algun exploit. Para lograr esto debemos concatenar los opcodes en una cadena de caracteres anteponiendoles 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 llamamos a ejecutar la cadena de caracteres dentro de nuestro código en C, así que nuestro 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 nuestro shellcode se ejecute correctamente compilamos nuestro código en C usando GCC y lo ejecutamos:

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

Perfecto, ahora hemos creado un shellcode funcional que imprime la cadena 'hola'. En la próxima entrega descubriremos cómo optimizar este shellcode para hacerla aún mas pequeña y además crearemos una shellcode que haga honor a su nombre, esto es por supuesto, ejecutar una shell. ¡Hasta la próxima!

End of file.