Bisoños Usuarios de GNU/Linux de Mallorca y Alrededores | Bergantells Usuaris de GNU/Linux de Mallorca i Afegitons

Fundamentos de los *Buffer Overflow*'s (29707 lectures)

Per **Eduard Llull**, ***Daneel*** ()

Creado el 08/06/2001 18:32 modificado el 08/06/2001 18:32

Seguro que muchos de vosotros habeis oido hablar de los desbordamientos de buffer o buffer overflow, fuente de muchos exploits, pero no sabeis como funcionan. En este artículo explicaré sus fundamentos, aunque no os penseis que os voy a dar la receta para hackear a nadie 8-).

Empezaremos recalcando que el lenguaje C no comprueba en tiempo de ejecución los limites de los arrays. Para demostrarlo utilizaremos un sencillo ejemplo:

```
// File: ej1.c
#include
#include

int main(int argc, char *argv[]) {
  char arr1[5] = "AAAA\0";
  char arr2[5];

  puts(arr1);
  strcpy(arr2, "BBBBBBBBBBB");
  puts(arr1);

  exit(0);
}
```

Si ejecutamos este simple programa veremos que en el segundo `puts(arr1)`, la salida es 'BBBB'. En el strcpy, el string origen es más largo que el destino (arr2) por lo que ha empezado a sobreescribir posiciones de memoria (arr1).

Otro concepto que debemos aclarar, antes de meternos de lleno en la explicación de los desbordamientos de buffer, es el funcionamiento de la pila. Para los que no lo sepan, una pila es una cola FILO (First In, Last Out). Utilizando el simil que usaron cuando me lo explicaron a mí, podemos pensar en la pila como un montón de platos: cuando queremos poner un plato al montón lo ponemos encima, y para quitar el que está debajo debemos quirtar primero los de encima. Pues justamente así funciona la pila, lo primero que se introduce en la pila, es lo último que sale. Para introducir datos en la pila contamos con la instrucción *push*, mientras que para sacarlos tenemos *pop*
Para saber en que posición de memoria se encuentrá lo más alto de la pila, la arquitectura ix86 cuenta con el registro %esp (*Stack Pointer*). Remarcaremos que en esta arquitectura la pila crece hacia las posiciones de memória más bajas, es decir, cuanto más cosas metamos en la pila, más baja será la dirección de la memoria en que lo escribamos.Cada una de las posiciones de la pila es una longword (4 bytes).

Muy bien, ahora veamos que ocurre en nuestra CPU cuando llamamos a una función. Para entender esta parte serán necesarios tener ciertas nociones del funcionamiento interno de los microprocesadores. Utilizaremos un simple programa y el debugger gdb:

```
// File: ej2.c
#include
#include
```

```
void funcion(char *p) {
  char arr[16];
  strcpy(arr, p);
  return;
}

int main(int argc, char *argv[]) {
  funcion(argv[1]);
}
```

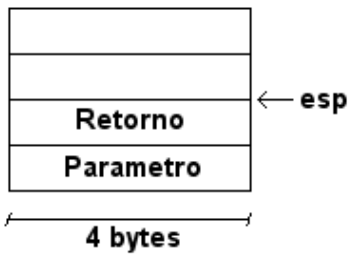
A continuación ejecuteremos:

```
$ cc ej2.c -o ej2
$ gdb ej2
(gdb) disassemble main
Dump of assembler code for function main:
0x80483ec <main>:       push   %ebp
0x80483ed <main+1>:     mov    %esp,%ebp
0x80483ef <main+3>:     mov    0xc(%ebp),%eax
0x80483f2 <main+6>:     add    $0x4,%eax
0x80483f5 <main+9>:     mov    (%eax),%edx

0x80483f7 <main+11>:    push   %edx
0x80483f8 <main+12>:    call   0x80483d0
0x80483fd <main+17>:    add    $0x4,%esp

0x8048400 <main+20>:    leave
0x8048401 <main+21>:    ret
0x8048402 <main+22>:    nop
...
End of assembler dump.
```

En rojo tenemos remarcado lo que nos interesa: metemos el parámetro de la función en la pila, llamamos a la función y cuando regresamos, recuperamos el espacio de la pila utilizado por el parametro. Pero ¿cómo sabe la función a que dirección debe volver cuando acabe? Para resolver este problema, la propia instrución *call* introduce en la pila la dirección de retorno. Entonces, en el momento en que llamamos a la función tendríamos la pila como:



Veamos ahora que hace la función:

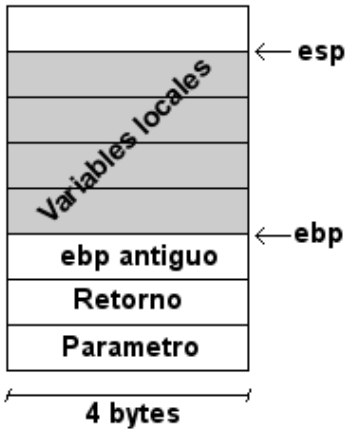
```
(gdb) disassemble funcion
Dump of assembler code for function funcion:

0x80483d0 <funcion>:    push   %ebp
0x80483d1 <funcion+1>:  mov    %esp,%ebp
0x80483d3 <funcion+3>:  sub    $0x10,%esp

0x80483d6 <funcion+6>:  mov    0x8(%ebp),%eax
0x80483d9 <funcion+9>:  push   %eax
```

```
0x80483da <funcion+10>: lea    0xfffffff0(%ebp),%eax
0x80483dd <funcion+13>: push   %eax
0x80483de <funcion+14>: call   0x8048308
0x80483e3 <funcion+19>: add    $0x8,%esp
0x80483e6 <funcion+22>: jmp    0x80483e8
0x80483e8 <funcion+24>: leave
0x80483e9 <funcion+25>: ret
0x80483ea <funcion+26>: mov    %esi,%esi
End of assembler dump.
```

Lo primero que hace es guardar el registro %ebp en la pila y después copia el valor de %esp a %ebp. El registro %ebp se utiliza para poder referenciar de forma sencilla a los parámetros que fueron pasados a la función. Después, restamos 16(0x10) bytes al puntero de la pila. Recordad que la pila crece hacia la posición de memoria 0. Con esta resta hemos reservado memoria para las variables locales de la función. En este momento el estado de la pila será:



Los desbordamientos de buffer se basan en introducir código máquina en el espacio reservado para las variables locales y después modificar la dirección de retorno para que apunte a la posición de memoria donde hemos introducido nuestro código. Este código puede ser lo que se conoce como un *shell code*, es decir, las instrucciones necesarias para lanzar un interprete de comandos como puedan ser el bash, sh, etc.
Lo que se suele hacer es aprovechar la función **strcpy** para sobreescribir posiciones de memoria. El string origen de la función **strcpy** contiene nuestro código máquina al principio y después la nueva dirección de retorno repetida tantas veces como sea necesaria para llegar a sobreescribir la antigua. De esta manera, cuando la función llegue a su fin y llame a la instrucción *ret*, que recupera la dirección de retorno de la pila, la ejecución del programa continuará en la posición de memória que nosotros le hemos indicado y en la que tenemos nuestro código.

Bueno, espero que este artículo hos haya ayudado a comprender como funcionan los archiconocidos y temidos *buffer overflows*. Para acabar diré que hay parches para el kernel de Linux que evitan que se pueda ejecutar código máquina que resida en la pila. De esta manera se imposibilitan este tipo de ataques.

E-mail del autor: daneel _ARROBA_ bulma.net
Podrás encontrar este artículo e información adicional en: http://bulma.net/body.phtml?nIdNoticia=660