Creación de Shellcodes para Exploits en Linux/x86

Daniel Fdez. Bleda
dfernandez@isecauditors.com
Internet Security Auditors, S.L.

HackMeeting III
(Madrid, 5 octubre 2002)

Índice (I)

Parte TEÓRICA.

- Un poco de historia.
- Conceptos básicos: registros, memoria, ASM, etc.
- Tipos de exploits.
- Modo de actuación de cada tipo de exploit.
- Sistemas de protección “anti-exploits”.
- Métodos de evasión anti protectores de pila/heap.
- Programación de shellcodes para exploits.
- Métodos de evasión en shellcodes anti IDS.

Índice (II)

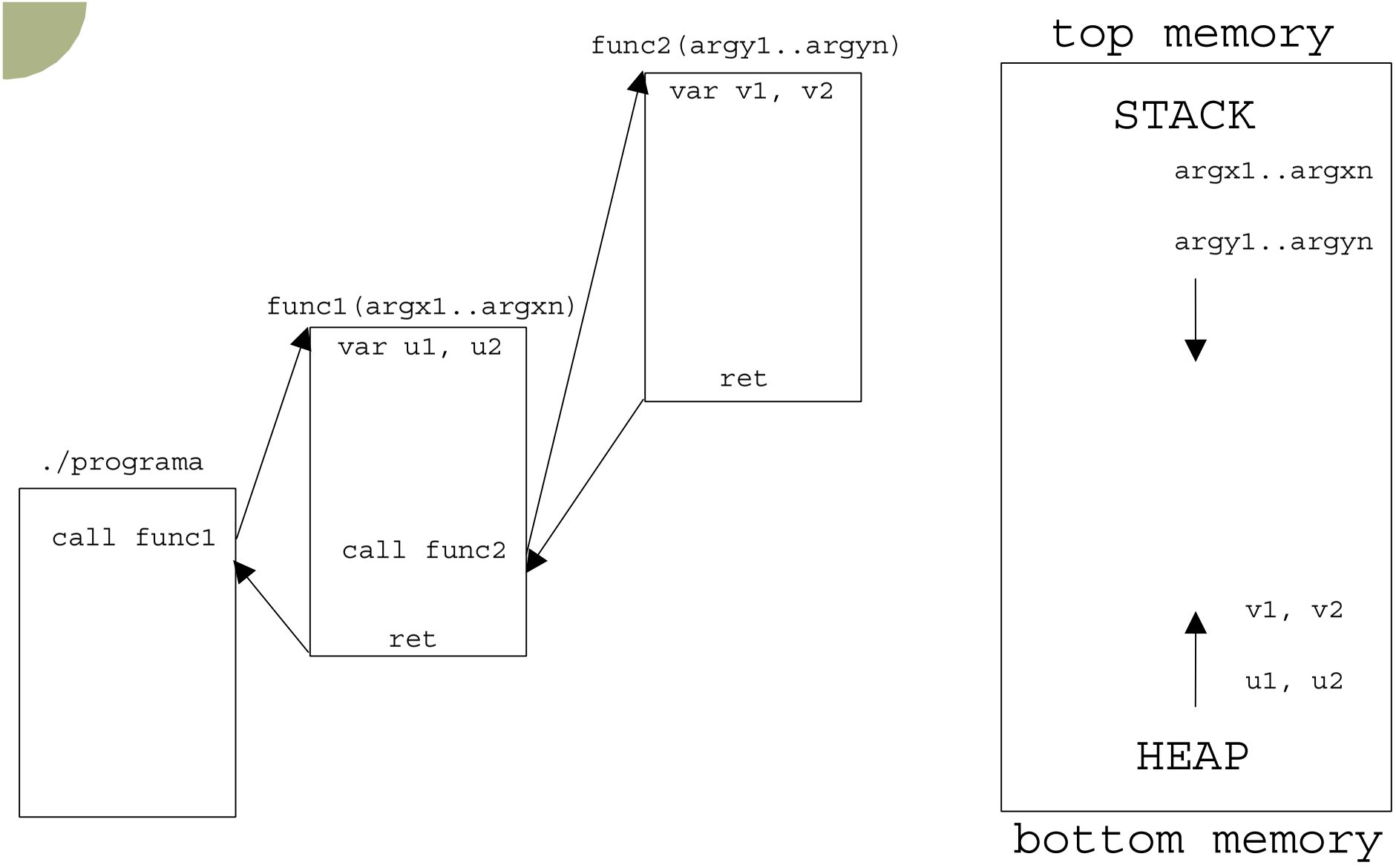
Parte PRÁCTICA.

- Funcionamiento básico de exploits:
  - Stack.
  - Heap (pointer/function pointer/VPTR overwrite).
  - Format String.
- Casos de reales:
  - Remote root con un b0f.
  - Remote root con un exploit de formato (Posadis DNS, Washington University (wu-ftp) FTP).

El primer Exploit

- 2 de noviembre de 1988, un gusano, el "Internet Worm", causa los mayores daños de la historia, hasta entonces...
- b0f en fingerd y replicación con sendmail.
- Desde entonces, se han explotado buffer overflows en todo tipo de aplicaciones y en todo tipo de sistemas operativos.

La Memoría de un Proceso



Conceptos de ASM (I):
Registros

Registro	Nombre	Funciones
a	Acumulador	Tipo syscall
		Valor de retorno
		Funciones I/O
b	Base	Parametro 1°
		Valor de retorno
c	Contador	Contador en bucles
		Parámetro 2°
		Valor de retorno
d	Datos	I/O a puertos
		Parámetro 3°
		Aritméticas
		Syscalls
esp	Frame Pointer	Top frame actual de la pila
ebp	Stack Pointer	Base frame actual de la pil
eip y edi	Segmentos	Paso de datos en el scode

- e*x (32b), *x (16b), *h (8b + sig), *l (8b – sig).

Conceptos de ASM (I):
Instrucciones y tipo de datos

Instrucción	Función
mov	mover datos entre/de/a un registro/memoria
inc	incrementar un registro
dec	decrementar un registro
add	sumar algo a/desde un registro
sub	restar algo a/desde un registro
xor	0 xor 0=0;0 xor 1=1;1 xor 0=1;1 xor 1 = 0;
lea	pasar un dato en mem a un registro (load effective address)
int	ejecuta una interrupción: syscall(0x80)
push	añade un elemento en la pila
pop	extrae un elemento de la pila

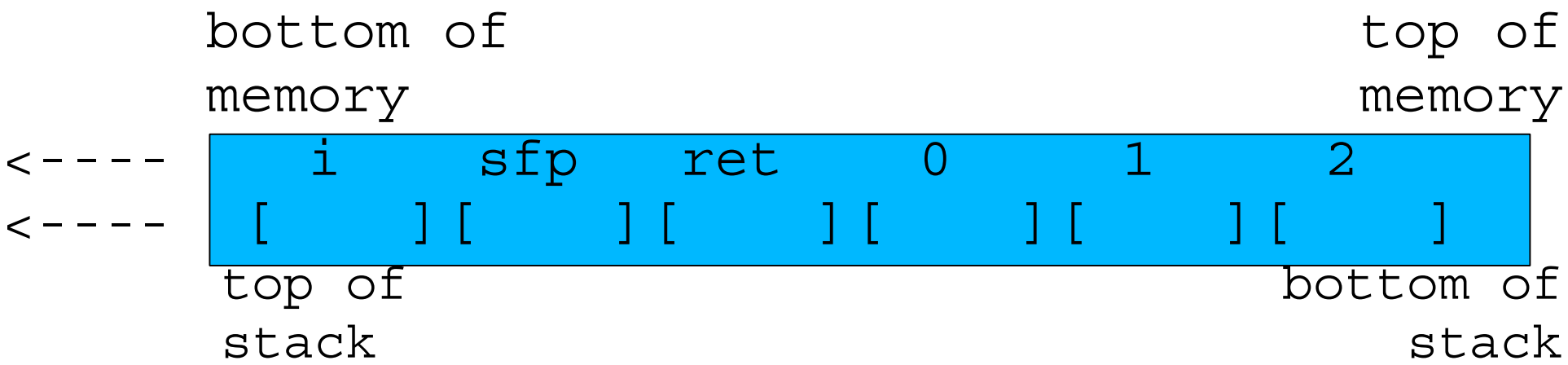
La Pila

- Parámetros en llamadas a funciones.
- Variables locales.
- Valores de retorno.
- Permitir controlar el flujo de ejecución tras una llamada a función.

La pila en una llamada a función

```
int funcion(int x, int y, int z)
{
        int i = 4;
        return (x + i);
}

int main()
{
        funcion(0, 1, 2);
        return 0;
}
```



Explotando la pila(I):
Cambiando el flujo de ejecución

- Si podemos modificar la dirección de retorno (ret) guardada de forma implicita por “call” podremos variar el flujo norma de ejecución.
- ¡x = 1 = 0 !

```
Void funcion(int a, int b, int c)
{
        char buffer1[5];
        char buffer2[10];
        int *ret;

        ret = buffer1 + 12;
        (*ret) += 8;
}
int main()
{
        int x;

        x = 0;
        funcion(1, 2, 3);
        x = 1;

        printf("%d\n",x);
}
```

Heap

- Espacio para variables que emplean memoria de forma dinámica.
- El espacio es reservado por la aplicación.

Explotando el Heap (I):
Generalidades

- Es más difícil de conseguir que el stack overflow.
- Basado en técnicas diversas:
  - Sobreescritura de apuntadores a funciones.
  - Sobreescritura de Vtables.
  - Explotación de librerías malloc.
- Requiere condiciones concernientes a la organización de un proceso en la memoria.

Explotando el Heap (II):
Sobreescritura de punteros

- Requiere un orden estricto en la declaración de variables:

```
...
static char buf[BUFSIZE];
static char *ptr;
...
```

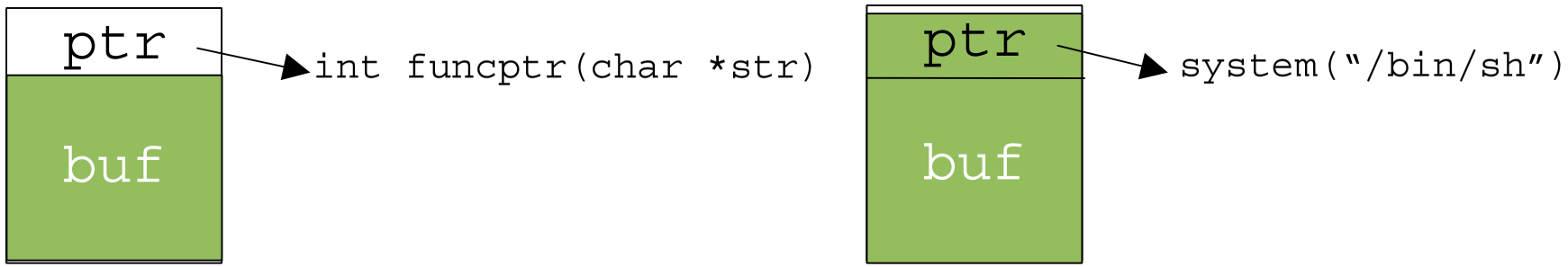
- Es difícil que se produzca esta condición.
- Requiere localizar la dirección de argv[1].
- Es independiente del SO.

Explotando el Heap (II):
Sobreescritura de punteros a funciones

- Requiere un orden estricto en la declaración de variables:

```
...
static char buf[BUFSIZE];
static int (*funcptr)(const char *str);
...
```

- Es difícil que se produzca esta condición.
- Permite ejecutar otra función o shellcode.



Explotando el Heap (IV):
Alterando VPTR Tables (C++) - 1

- Explota el mecanismo dinámico de las llamadas a funciones virtuales en una clase.
- Pocas aplicaciones se desarrollan en C++, con lo que es complicado de explotar.
- Precondición: Necesita que se declare un buffer y una función virtual dentro de la clase a explotar.

Explotando el Heap (V):
Alterando VPTR Tables (C++) - 2

```
class BaseClass
{
    private:
        char Buffer[100];

    public:
        void SetBuffer(char *String) {
        strcpy(Buffer,String);
        }
        virtual void PrintBuffer() {
            printf("%s\n",Buffer);
        }
};
```

```
class MyClass1:public BaseClass
{
    public:
        void PrintBuffer()
        {
        printf("MyClass1: ");
          BaseClass::PrintBuffer();
            }
};
```

```
B son los bytes Buffer
V el byte de la VPTR a VTABLE_MyClass1
I es información de la clase que
  hereda de BaseClass (MyClass1)
P es el apuntador a la fun. PrintBuffer
```

```
Object[0]: BBBBBB....BBBBBBBVVVV
                                =+==
          +--------------------+
          +--> IIIIIIIIIIIIIPPPP
               VTABLE_MyClass1
```

```
+----(1)---<------------------+
|                          ==+=
SSSS..SSSS..NNN..CCCCC..CCCCVVVV0
|                 +
+----(2)--->-----+
```

```
S es la dir. de inicio del shcode
N bytes de alineamiento (NOPs)
C codigo del shellcode
V Puntero a la función virtual
0 0x00h que finaliza el buffer
```

Exploits de Format String (I)

- Emplea una técnica reciente (1999), pero explota vulnerabilidades que existían desde hace años (de 2 a 6 y hasta 8).
- Explota la posibilidad de poder pasar directamente a una funcion *printf un parámetro no parseado: `printf(input)`.
- Aprovecha bugs de programación fáciles de encontrar.
- *write-anything-anywhere*.

Exploits de Format String (II)

- Emplean la capacidad de poder desplazarse por la memoria definiendo longitudes de numeros en una cadena de formato (%Nx).
- Y la utilidad de una opción, %n, que devuelve la cantidad de bytes escritos previamente.
- Desplazandonos por la memoria y sabiendo la posición de memoria a escribir y descomponiéndola mediante %hn, podemos escribir word a word, esta posición.

Shellcodes (I)

- Es un conjunto de instrucciones compiladas de ensamblador que realizan una función normalmente simple y con unas restricciones precisas debido a su uso.
- Toma el nombre a su primer objetivo: ejecutar una *shell.*

Shellcodes (II):
Herramientas

- gdb: debug/trace.
- nasm: compilación de códido ensamblador.
- disasm: desemblado/análisis de scodes.
- Un ejecutor de shellcodes.
- Un convertidor de scodes a binarios para análisis de shellcodes.

Shellcodes (II):
Limitaciones

- NULL byte: No pueden contener **\00** (eos).
- Addressing problem: No se pueden emplear dirección de memoria *hardcoded.*
- Tamaño: No se suele disponer de buffers muy grandes. El shellcode ha de ser reducido.

Shellcodes (IV):
Métodos de evasión anti IDS

- Ofuscación de cadenas sensibles a la detección (p.e. “/bin/sh”).
- Shellcodes alfanúmericos (Raise (netsearch), rix (Phrack)).
- Shellcodes comprimidos.

Sistemas de protección "Anti-exploits": Introducción

- La mayoría de exploits son posibles gracias a funciones C no fiables (e.j strcpy, sprintf, ..).
- Dos mecanismos de protección: Libsafe, Grsecurity, StackGuard y StackShield.

Sistemas de protección "Anti-exploits": Libsafe (I)

- Librería que reescribe funciones sensibles de la librería libc (strcpy, strcat, sprintf, vsprintf, fscanf, scanf, sscanf,..).
- Lanza alertas en caso de detectar un posible intento de buffer overflow.
- Librería dinámica. Cargada en memoria antes que cualquier otra librería.
- Intercepta las llamadas a funciones peligrosas de libc y utiliza la suya en su lugar.
- Detecta violaciones en los límites de buffers
- Info: http://www.research.avayalabs.com/project/libsafe/

Sistemas de protección "Anti-exploits": Libsafe (II)

strcpy original:

Fin de bucle controlado únicamente por el fin de cadena '\0' !!!

```
char *strcpy (char *dest, const char *src)
{
    char *temp = dest;

    while ((*dest = *src)!= '\0')
    /*nothing*/

    return tmp;
}
```

strcpy de Libsafe:

max_size: distancia (#bytes) entre dest y el frame pointer de dest
=> tamaño máximo posible que puede tener dest.

```
char *strcpy (char *dest, const char *src)
{
    ...
    if ((len = strlen(src, max_size)) == max_size) /*overflow?*/
    _libsafe_die("Overflow caused by strcpy()");
    memcpy(dest,src,len+1); /* libreria estandard de C */

    return dest;
}
```

Sistemas de protección "Anti-exploits": Libsafe (III)

- Beneficios
  - Fácil de instalar. No se necesita recompilar el kernel
  - Buena o Mejor performance. Strcat de libsaf es más rápido que el original.
- Inconvenientes
  - Embedded Frame Pointer => gcc con -fomit-frame-pointer
  - Podemos ejecutar exploits basados en sobreescritura de punteros a ficheros o funciones (sin sobrepasar el max_size)

Sistemas de protección "Anti-exploits": GrSecurity

- Conjunto de parches para el kernel. Ofrecen la posibilidad de hacer las areas de memoria stack y heap no ejecutable.
- Open Wall -> Stack
- PaX -> Head y Stack
- www.grsecurity.net

Sistemas de protección "Anti-exploits": Open Wall (I)

- arch/i386/kernel/traps.c
- Segmentation fault: Cuando sucede un evento de memoria inesperado, se procesa en la función do_general_protection.
- do_general_protection: Si estado de los registros no refleja ninguna de las situaciones previstas => Open Wall ofrece nuevas posibilidades de analisis.
- Si alguna de los analisis proporcionados por Open Wall detecta un intento de ejecutar instrucciones en la pila => Se lanza una alerta

Sistemas de protección "Anti-exploits": Open Wall (II)

```
asmlinkage void do_general_protection(struct
  pt_regs * regs, long error_code)
{
#ifdef CONFIG_GRKERNSEC_STACK
  unsigned long addr;
#ifdef CONFIG_GRKERNSEC_STACK_GCC
  unsigned char insn;
  int err, count;
#endif
#endif
        if (regs->eflags & VM_MASK)
                goto gp_in_vm86;

        if (!(regs->xcs & 3))
                goto gp_in_kernel;

#ifdef CONFIG_GRKERNSEC_STACK
.....
/* * * Check if we are returning to the stack
  area, which is only likely to happen * * when
  attempting to exploit a buffer overflow. * */

if ((addr & 0xFF800000) == 0xBF800000 || (addr
  >= PAGE_OFFSET - _STK_LIM && addr <
  PAGE_OFFSET))
```

```
security_alert("return onto stack by "
  DEFAULTSECMSG, "returns onto stack",
  DEFAULTSECARGS);
...

  Current->thread.error_code = error_code;
        current->thread.trap_no = 13;
        force_sig(SIGSEGV, current);
        return;

gp_in_vm86:
        handle_vm86_fault((struct
  kernel_vm86_regs *) regs, error_code);
        return;

gp_in_kernel:
        {
                unsigned long fixup;
                fixup =
  search_exception_table(regs->eip);
                if (fixup) {
                        regs->eip = fixup;
                        return;
                }
                die("general protection
  fault", regs, error_code);
        }
}
```

Sistemas de protección "Anti-exploits": PaX (I)

- Utiliza los mecanismos de paginación (PTE, DTLB, ITLB): Cuando se produce un fallo de página la CPU carga la nueva página utilizando el PTE, que contiene los permisos para cada página.
- Controla las páginas de memoria ejecutables mediante un sistema de estados y transiciones.
- Implementa nuevas funcionalidades en el mecanismo para el control de los fallos de página => arch/i386/mm/fault.c

Sistemas de protección "Anti-exploits": PaX (II)

- Inconvenientes de no permitir la ejecución de código en la pila o el heap:
  - Algunas aplicaciones no funcionan: Servidores XFree86-4.
  - Lenguajes como Java tiene VM que requieren un stack ejecutable.

Sistemas de protección "Anti-exploits": PaX (III)

- Return into libc
  - Evasión para PaX o Open Wall
  - No ejecuta código en la pila o heap -> llamada a una función de la librería libc (system()). Ret = @función a llamar
  - Sólo necesitamos saber @función a llamar.
  - Solución PaX: Cambiar la dirección de la función cada vez que se llama => mmap randomization => Fuerza Bruta

Sistemas de protección "Anti-exploits": StackGuard

- Técnica de compilación.
- "Permite eliminar stack smashing attacks".
- "canary" cerca de la dirección de retorno. Si canary ha sido alterando cuando la función retorno => intento de ataque => alerta.
- Canary Spoofing: Terminator (NULL, CR, LF, EOF), Random, etc.

Sistemas de protección "Anti-exploits": StackShield

- Utilidad que permite añadir protección en tiempo de compilación.
- Solo controlamos el código que compilamos nosotros. No podemos controlar binarios que instalamos por ejemplo mediante paquetes RPM.
- www.angelfire.com/sk/stackshield/

Bibliografía (I):
Stack Overflows

- prym: “finding and exploiting programs with buffer overflows”.
  http://destroy.net/machines/security/buffer.txt
- lefty: “Buffer overruns, whats the real story".
  http://reality.sgi.com/nate/machines/security/stack.nfo.txt
- kekabron: “Buffer Overflows (b0f’s)”. Netsearch Ezine #4 (0x11).
  http://www.netsearch-ezine.com
- klog: “The Frame Pointer Overwrite". Phrack #55 (0x08).
  http://www.phrack.org/show.php?p=55&a=8
- Aleph1: “Smashing the Stack for Fun and Profit”. Phrack #49 (0x05).
  http://www.phrack.org/show.php?p=56&a=5

Bibliografía (II):
Heap Overflows

- Fayolle, Pierre-Alain; Glaume, Fayolle: “A Buffer Overflow Study. Attacks & Defenses".

  http://www.enseirb.fr/~glaume/bof/report.html

- Matt Conover & w00w00 Security Team: “w00w00 on Heap Overflow".

  http://www.w00w00.org/articles.html

- cafo: “Heaps Overflows (1/2)”. Netsearch Ezine #4 (0x14).

  http://www.netsearch-ezine.com

- rix: “Smashing C++ VPTRS". Phrack 56 (0x08).

  http://www.phrack.org/show.php?p=56&a=8

- twitch: “Taking advantage of non-terminated adjacent Memory Spaces”. Phrack 56 (0x0e).

  http://www.phrack.org/show.php?p=56&a=14

Bibliografía (III):
String Format exploits

- gera & riq: "Advances in format string exploiting". Phrack 59 (0x07).
  http://www.phrack.org/show.php?p=59&a=7
- scut/team teso: "Exploiting Format String Vulnerabilities".
  http://teso.scene.at/releases/formatstring-1.2.tar.gz
- Umesh Shankar: “Detecting Format String Vulnerabilities with Type Qualifiers”.
  http://qb0x.net/papers/FormatString/usenix01/usenix01.pdf
- RaiSe: “Bugs de Formato (1/2)”. NetSearch Ezine.
  http://www.netsearch-ezine.com
- The Itch: “Exploiting local format string holes on x86/ linux”.
  http://qb0x.net/papers/FormatString/fmtpaper.txt
- Frédéric Raynal: “Howto remotely and automatically exploit a format bug”.
  http://www.security-labs.org/cvRaynal.pdf

Bibliografía (IV):
Writing Shellcodes (I)

- Fayolle, Pierre-Alain; Glaume, Fayolle: “A Buffer Overflow Study. Attacks & Defenses".

  http://www.enseirb.fr/~glaume/bof/report.html

- Matt Conover & w00w00 Security Team: “w00w00 on Heap Overflow".

  http://www.w00w00.org/articles.html

- Umesh Shankar: “Detecting Format String Vulnerabilities with Type Qualifiers”.

  http://qb0x.net/papers/FormatString/usenix01/usenix01.pdf

- RaiSe: “Shellcodes en Linux/i386”. Netsearch Ezine #4 (0x04).

  http://www.netsearch-ezine.com

- RaiSe: “Shellcodes en Linux/i386 (2)”. Netsearch Ezine #5 (0x04).

  http://www.netsearch-ezine.com

- The Last Stage of Delirium Research Group: “UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes”. Version 1.0.2.

  http://lsd-pl.net/papers.html#assembly

Bibliografía (IV):
Writing Shellcodes (II)

- Zillion: “Writing Shellcode”.
  https://uhf.ath.cx/papers/Writing_shellcode.htm
- Miyagi, Robin: “Linux Assembler Tutorial”.
  http://www.geocities.com/SiliconValley/Ridge/2544
- “Linux System Call Table”.
  http://quaff.port5.com/syscall_list.html
- UNF && pr1: “Writing Linux/x86 shellcodes for dum dums”.
  http://www.u-n-f.com/papers/shellcode-pr10n.txt

Bibliografía (V):
Sistemas de protección/evasión.

- Bulba and Kil3r: “Bypassing StackGuard and StackShield”. Phrack 56 (0x05).

  http://www.phrack.org/show.php?p=56&a=5

- “Bypassing PaX ASLR Protection”. Phrack 59 (0x09).

  http://www.phrack.org/show.php?p=59&a=9

- Nergal: “The advanced return-into-lib(c) exploits (PaX case study)”. Phrack 58 (0x04).

  http://www.phrack.org/show.php?p=58&a=4

- Cowan, Crispin; Wagle, Perry; Pu, Calton; Beattie, Steve; Walpole, Jonathan: “Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade”

  http://www.cse.ogi.edu/DISC/projects/immunix

- Fayolle, Pierre-Alain; Glaume, Fayolle: “A Buffer Overflow Study. Attacks & Defenses".

  http://www.enseirb.fr/~glaume/bof/report.html