



**Título:** **Crackeando apps para PocketPC / Windows Mobile**

**Autor:** nap

**Temática:** ingeniería inversa

**Plataforma:** Windows Mobile

**Fecha:** 30/04/2008

**Licencia:** Este documento es propiedad de Disidents, su distribución/modificación siempre está limitada a la autorización explícita del autor o de la organización. En general siempre que no se modifique el contenido ni se elimine esta cabecera la distribución está autorizada, salvo que Disidents o el autor indiquen lo contrario.

**Fuente:** <http://www.disidents.org>

**Ezine:** Disidents 9

**1.0- Primeros pasos**

**2.0-Desensamblado**

**3.0- Crack**

**4.0- Construyendo y comprendiendo el código**

**5.0- Ejecutar en el dispositivo**

## 1.0 - Primeros pasos

Es necesario tener un desensamblador que soporte el procesador de tu dispositivo ([IDA Pro](#) es el más recomendable), un editor hexadecimal y una app que te apetezca crackear.

Una vez hemos instalado la aplicación nos pasamos el exe a nuestro PC y lo desensamblamos con IDA. Doy por hecho que el lector tiene ya algo de experiencia en crackear en otras plataformas.

Los procesadores más comunes en smartphones/PDAs con Windows Mobile son ARM, y es lo con lo que vamos a estar trabajando en este caso. Su lenguaje ensamblador es relativamente sencillo y es fácil construir las instrucciones que se quiera a mano, además tienen longitud fija (4 bytes). Es recomendable leerse algunos manuales y documentos de introducción si no se tiene experiencia con él:

[http://www.simplemachines.it/doc/arm\\_inst.pdf](http://www.simplemachines.it/doc/arm_inst.pdf)

<http://www.arm.com/miscPDFs/14128.pdf?>

## 2.0 - Desensamblado

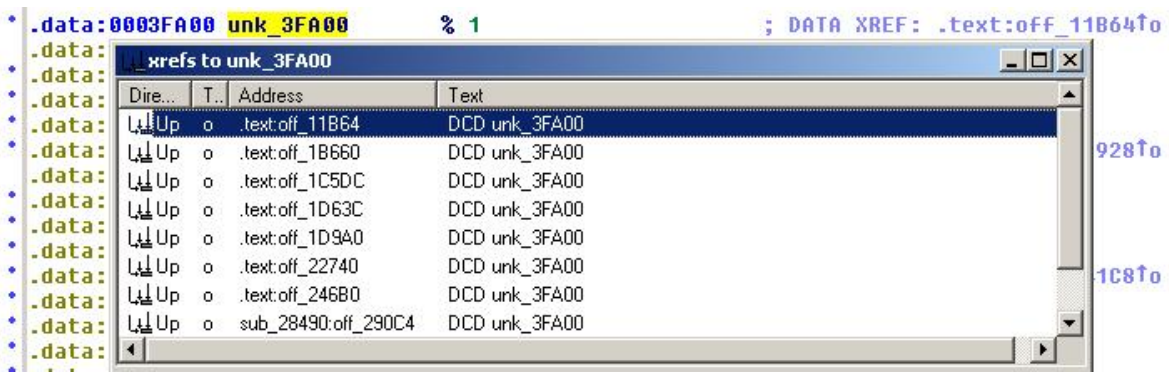
Como ejemplo vamos a crackear una app cualquiera cuyo nombre no vamos a revelar, no vayáis a hacer algo ilegal ;)

Primero buscamos con el IDA algunos strings relevantes que hemos visto al ejecutarla, como “unregistered” en el about box, o una mención al periodo de trial nada más abrirla y tras buscar un poco llegamos a esto:

```
.text:00011A68      LDR     R3, =unk_3FA00
.text:00011A6C      LDR     R3, [R3]
.text:00011A70      CMP     R3, #0
.text:00011A74      BEQ     loc_11AE0

.text:00011AE0 ; -----
.text:00011AE0      ; CODE XREF: sub_118DC+198↑j
.text:00011AE0      loc_11AE0      ; CODE XREF: sub_118DC+198↑j
.text:00011AE0      LDR     R2, =aUnregisteredVe ; lpString
.text:00011AE4      LDR     R0, [R4]              ; h01n
.text:00011AE8      ORR     R1, R7, #0xFFaUnregisteredVe unicode 0, <Unregistered Version>,0
.text:00011AEC      BL      SetDlgItemTextW
```

Como podemos ver, se carga en R3 el valor apuntado por el puntero en 3FA00 y se salta con la instrucción BEQ a 11AE0 en caso de que su valor sea cero, para poner “Unregistered version” en un elemento de un cuadro de diálogo. Parece un buen punto de partida. Ahora vamos a 3FA00, y comprobaremos las referencias que hay a esa dirección en toda la aplicación:



Vemos que hay unas cuantas referencias y las vamos recorriendo. Resulta que en todas o casi todas encontramos alusiones a cosas que tienen que ver con que la app este registrada o no, así que parece que hemos dado en el clavo.

### 3.0 - Crack

Entonces seguimos buscando en la lista hasta llegar a una distinta a las demás, porque no lee el contenido del puntero 3FA00, sino que lo escribe. Ahí es donde probablemente tengamos que hacer modificaciones para que el valor sea lo que nos interesa (distinto de 0, ya que, como hemos visto, 0 es su valor cuando la app no está registrada):

```

.text:00030EC8      LDR    R3, [R11,#var_204]
.text:00030ECC      LDR    R10, =unk_3FA00
.text:00030ED0      CMP    R3, #0
.text:00030ED4      BEQ    loc_30F68
.text:00030ED8      SUB    R0, R11, #0x290
.text:00030EDC      BL     sub_1EA24
.text:00030EE0      MOV    R1, R0
.text:00030EE4      SUB    R0, R11, #0x2A8
.text:00030EE8      BL     sub_197C0
.text:00030EEC      LDR    R3, [R11,#var_290]
.text:00030EF0      SUB    R4, R3, #0x10
.text:00030EF4      ADD    R0, R4, #0xC ; lpAddend
.text:00030EF8      BL     InterlockedDecrement
.text:00030EFC      CMP    R0, #0
.text:00030F00      LDRLE R0, [R4]
.text:00030F04      MOULE R1, R4
.text:00030F08      LDRLE R3, [R0]
.text:00030F0C      MOULE LR, PC
.text:00030F10      LDRLE PC, [R3,#4]
.text:00030F14      MOV    R3, #0x100
.text:00030F18      MOV    LR, #0
.text:00030F1C      ORR   R2, R3, #0x8E ; size_t
.text:00030F20      MOV    R1, #0 ; int
.text:00030F24      MOUL  R0, 0xFFFFFFFF23
.text:00030F28      MOV    R0, R0,LSL#1
.text:00030F2C      ADD    R0, R11, R0 ; void *
.text:00030F30      SUB    R12, R11, #0x100
.text:00030F34      STRH  LR, [R12,#-0xBC]
.text:00030F38      BL     memset
.text:00030F3C      LDR    R4, [R11,#var_2A8]
.text:00030F40      SUB    R0, R11, #0x1BC ; wchar_t *
.text:00030F44      MOV    R1, R4 ; wchar_t *
.text:00030F48      STR    R4, [R11,#var_2B0]
.text:00030F4C      BL     wcsncpy
.text:00030F50      SUB    R0, R11, #0x1BC ; lpsz
.text:00030F54      BL     CharLowerW
.text:00030F58      SUB    R1, R11, #0x218
.text:00030F5C      SUB    R0, R11, #0x1BC
.text:00030F60      BL     sub_1DA1C
.text:00030F64      STR    R0, [R10]
.text:00030F68      MOV    R2, #0x20
.text:00030F6C      MOV    R1, #0

```

```

LDR    R3, [R11,#var_204]
LDR    R10, =unk_3FA00
MOV    R3, #1
STR    R3, [R10]

```

→

```

BL     sub_1DA1C
MOV    R1, #1
MOV    R2, #0x20
STR    R1, [R10]

```

→

En la figura anterior podemos ver recuadrados en rojo dos posibles cambios. El primero bastaría por sí solo pero también se podría cambiar el BEQ por un NOP (en ARM no hay NOP propiamente, se pone *MOV R0, R0* como operación equivalente) y usar además el segundo cambio. Hemos preferido matar dos pájaros de un tiro con el primer cambio, anulando el salto y dando valor a lo que apunta 3FA00. Básicamente, estamos forzando que se almacene un 1 en la dirección apuntada por R10 (R10 había sido cargado previamente como 3FA00), de forma que luego cada vez que la aplicación compruebe el valor de lo que apunta 3FA00 encuentre un 1 y determine que está registrada correctamente. En el segundo cambio hacemos lo mismo, aprovechando un R1 que hemos comprobado que más adelante da igual que sea 0 ó 1.

#### 4.0 - Construyendo y comprendiendo el código

Las instrucciones típicas que vamos a estar modificando y creando son MOV, LDR, STR y Bxx principalmente. MOV sirve para mover valores en registros, LDR y STR para cargar y guardar registros de memoria y las instrucciones Bxx (branches) son saltos condicionales.

Antes de empezar, comprendamos como funcionan algunas cosas. Cuando en el desensamblado vemos #1 significa valor literal 1. Cuando vemos *LDR R10, =unk\_3FA00* significa que se carga en R10 el valor 003FA00; obviamente si las instrucciones son de 4 bytes, no puede haber esa dirección ahí, por lo que se almacena 003FA00 al final de la función y la operación LDR realmente usa una referencia relativa a ese valor almacenado. El desensamblador conoce esto y por eso pone un “=” delante de unk\_3FA00, para indicarnos que es como si en esa operación pudiéramos el *LDR R10, #3FA00* que no podemos poner normalmente por las limitaciones del procesador.

Por otro lado, los branches que estaréis parcheando serán saltos relativos (para saltos absolutos se hace un MOV directamente al program counter- PC), así que estad preparados para sacar la calculadora cuando sea necesario (pero en este ejemplo no es necesario). También conviene resaltar lo que significa BL (branch with link): es un salto con enlace al llamador, o sea, un call de un nivel de profundidad, no recursivo, que utiliza LR (link register) para almacenar la dirección de retorno. Para llamadas recursivas y a más profundidad, se puede usar también la pila para almacenar direcciones de retorno. Veréis por ejemplo cosas curiosas como *MOV PC, LR* para retornar de una función, o cómo una función que debería llamar a otra cuando termina, en realidad sólo salta a la siguiente, ya que así se ahorra retornar dos veces y puede hacer que la que retorne sea la segunda directamente sin tocar el valor de LR entremedias.

Respecto a construir las instrucciones, es bastante fácil teniendo el manual de ARM a mano, que especifica exactamente como se construyen. Incluso se puede hacer sólo mirando y comparando los bytes desensamblados en varias instrucciones, como se puede ver aquí:



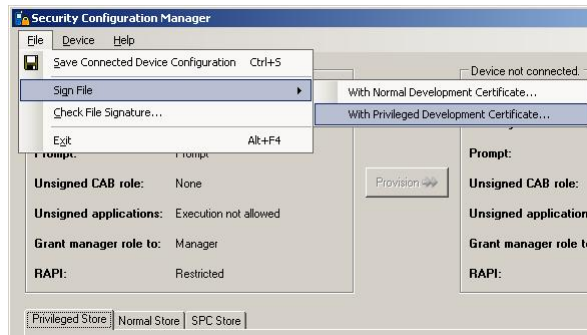
Están recuadradas en rojo las correspondencias entre el código fuente y los bytes de las instrucciones.

## 5.0 - Ejecutar en el dispositivo

Bueno, este último paso es una pequeña pesadilla la primera vez, porque no te esperas tener que firmar el ejecutable para poder ejecutarlo y además nadie ha pensado en el pobre cracker que va a tener que firmar el ejecutable directamente sin hacerlo como paso automático al generar el software con Visual Studio. Una vez te has enterado de que tienes que firmar la app y sabiendo como hacerlo, no tiene ningún misterio. Necesitas tener instalada alguna versión de VS o bajar independientemente la utilidad *signtool*.

Con los certificados adecuados y signtool se puede firmar la app, pero para no complicarnos vamos a usar una herramienta de Microsoft que además nos permite modificar la configuración de seguridad de nuestro dispositivo y algunas cosas más: [Device Security Manager PowerToy for Windows Mobile 5.0](#). Está pensada para usarse con VS 2005 y si no lo tenéis se quejará de que no encuentra *signtool*. En ese caso basta con que copiéis vuestro *signtool.exe* a la ruta que especifica (que es en la que estaría en VS2005) y listo.

Este es el menú que hay que usar en el *Device security configuration manager* para firmar aplicaciones:



Una vez hayáis firmado el .exe crackeado, ya podéis subirlo a vuestro dispositivo reemplazando el original y comprobar que la aplicación funciona correctamente.

---

Y bueno, esto es todo. Espero que os haya gustado y os aficionéis al tema ;) además conociendo ARM no sólo se pueden tocar apps para Windows Mobile, también hay todo tipo de dispositivos con microprocesadores ARM dentro, desde cámaras de fotos y video, cuya funcionalidad podéis trucar modificando el firmware, hasta routers y demás.