CRACKING
SIN SECRETOS
Ataque y defensa de software

Jakub Zemánek

Incluye CD-ROM

Ra-Ma®

Cracking sin secretos

Ataque y defensa de software

Jakub Zemánek

ÍNDICE

PRÓLOGO

Para la mayoría, el término cracking[1] contiene matices relacionados con lo prohibido, lo desconocido y lo extraño. ¿Por qué? ¿En qué consiste realmente el cracking? ¿Por qué resulta tan peligroso y qué defensas existen contra él? Entre otras cosas, este libro ayudará al lector a dar con la respuesta a estos interrogantes. La evolución del cracking ha ido siempre de la mano de la del anticracking. Estas dos técnicas no pueden sobrevivir la una sin la otra. No existiría protección de software sin al menos un conocimiento básico de las técnicas del cracking, y viceversa, no habría cracking sin un conocimiento mínimo sobre protección de software.

Por tanto, este libro muestra a los desarrolladores cómo proteger su software frente al cracking, así como las técnicas empleadas por los crackers[2]. Sólo de esta manera

---

[1] Nota del traductor: dada la ambivalencia con la que se aplica el término "cracking" a lo largo del libro (tanto para indicar una actividad relacionada con la protección del software como otra, ilegal, destinada a violar el código de un producto), la numerosa cantidad de técnicas que abarca (el autor lo considera una ciencia) y, sobre todo, la falta de equivalente satisfactorio en castellano que refleje esta complejidad semántica, se ha optado por dejarlo como un extranjerismo en el texto del libro, al igual que 'anticracking'.

[2] Nota del traductor: si bien a priori el lector puede tender a pensar que la mayoría de los "crackers" son, sencillamente, infractores, el propio uso que hace el autor del término 'cracker' (y que algo más adelante en este mismo prólogo aclara) le despoja de todo matiz moral. No existe voz equivalente en castellano ni siquiera aproximada.

conocerá el programador el tipo de amenazas que se ciernen sobre sus aplicaciones y a probarlas antes por sí mismo. Se aplica así el viejo principio del arte de la guerra: conoce a tu enemigo.

Este libro no sólo muestra los conceptos básicos, sino que también recoge algunas técnicas avanzadas de cracking y anticracking, además de una gran cantidad de información: desde la simple descripción de algoritmos de protección hasta el proceso de creación de un codificador PE propio. Si bien los profesionales del campo lo encuentren valioso, el libro no está dirigido a ellos en absoluto. Se ha escrito principalmente para principiantes, quienes deben de poseer los fundamentos mínimos de los lenguajes de programación.

A lo largo del libro se presenta una gran cantidad de código escrito en ensamblador. Que no asuste a nadie. Todo el mundo sabe que el ensamblador no es precisamente el mejor lenguaje de programación para principiantes. Por esa razón, todos los fragmentos del código están comentados[3]; así, cada operación resultará obvia incluso para quienes no sepan ensamblador en absoluto. Indirectamente el lector también obtendrá con este libro cierto conocimiento de ensamblador. Con ánimo de facilitar su lectura, se incluye un capítulo de referencia con la descripción de las instrucciones básicas de este lenguaje. No obstante, considérese que el ensamblador constituye un lenguaje de programación muy útil y muy eficaz que, aunque no sea un requisito para entender el texto, sí debería considerarse seriamente aprenderlo.

En todo caso, ¿por qué se emplea ensamblador? La respuesta es bien sencilla: los lenguajes de programación de alto nivel resultan poco ágiles y sus posibilidades, muy escasas a la hora de programar las protecciones de software. Razón por la que el autor ha decidido emplear código ensamblador con lenguajes de programación de alto nivel. Con ello se facilita su uso en lenguajes populares como C++ o Delphi además de poder portar el código (en este caso se ha empleado el compilador Microsoft Visual C++ 6.0). De esta manera, los desarrolladores podrán cortar y pegar fácilmente el código fuente necesario para incluirlo en sus proyectos.

Tras un largo periodo de dedicación a este campo, se puede afirmar que no existe una protección invencible. Ahora bien, sí es posible crear una protección de software resistente a la mayoría de las técnicas de ataque cuya anulación además acarree una gran cantidad de tiempo. Precisamente es su duración el factor crucial que hace satisfactoria una protección. Una protección satisfactoria puede definirse como la que resiste la mayor cantidad de tiempo. Por otra parte, siempre existirán infractores a quienes no les importe pasarse horas, días o incluso semanas intentando anular una protección. En la mayoría de

---

[3] Nota del traductor: y traducidos. También se han traducido los mensajes y literales de los programas siempre que con ello no quedara alterada la lógica del código.

las ocasiones, no se trata de una cuestión económica o de índole similar sino su deseo de vencer, de ser el mejor. Representa su entretenimiento, su afición. Por otro lado, hay empresas de software para quienes el tiempo significa dinero. Cuanto más tiempo lleve anular una protección, más ingresos obtendrá la empresa por ventas. Mucha gente, en vez de quedarse esperando a una versión pirata o a un crack válido, seguramente comprarán el programa original, lo que redundará en beneficio de las empresas, ¡siempre y cuando el programa ilegal no esté disponible antes que el original! El período inmediatamente posterior a su comercialización se revela casi siempre crucial. Resulta sumamente comprometido el que una versión pirata de un programa que ha suscitado gran expectación, se encuentre en Internet horas después de que se ponga a la venta.

Antes de adentrarse en profundidad en el mundo del cracking conviene realizar algunas puntualizaciones sobre la organización de este libro. Puesto que no existen límites definidos entre las distintas técnicas del cracking y la mayoría de ellas se complementan entre sí, muchos de los capítulos de este libro podrían reorganizarse de otra manera. Se ha ordenado según la frecuencia en el uso de la información que contienen y no según ningún otro criterio.

¿En qué consiste el cracking?

DEFINICIÓN DE CRACKING

El cracking se puede describir como el grupo de técnicas empleadas para codificar, analizar y estudiar los principios de un programa sin disponer de su código fuente. Con un caso práctico quedará mucho más clara esta breve definición. Cuando un desarrollador crea un programa, comienza escribiendo el código fuente en el lenguaje de programación que haya elegido para acabar compilándolo (en un programa ejecutable). Llegado este punto, nadie podría editar el programa sin disponer del código fuente y realizar una nueva compilación. Esto es falso, y en ello es precisamente en lo que se basa el cracking.

La técnica de ingeniería inversa constituye la piedra angular del cracking, se basa en la descompilación, o compilación inversa, de un programa a un lenguaje de programación, generalmente, el más básico, esto es, ensamblador. Existen descompiladores capaces también de descompilar un programa a un lenguaje de programación de alto nivel. No obstante, no se han recogido en este libro por resultar problemáticos, y faltos de la fiabilidad y de la exactitud requeridas en la práctica.

Aunque parezca que el objetivo principal del cracking consista en alterar el software con ciertas modificaciones de su funcionalidad original (normalmente cambios relativos a la seguridad o a las propiedades de la protección), excede la sencilla actividad de editar el código del programa. Puede llegar a anularse una buena parte de los sistemas de protección sin practicar ninguna modificación al programa: hallando la contraseña, el número de registro, etc., e incluso simplemente estudiando el código de programa.

Lo que importa verdaderamente a un cracker es un buen conocimiento de ensamblador. Exagerando y forzando un poco esta afirmación, podría decirse que todo programador en ensamblador representa un cracker en potencia, y viceversa. Lo que queda demostrado por la mala interpretación y error que cometen muchas personas siempre dispuestas a denunciar como criminal a cualquier cracker (puede que la causa resida en cierto número de películas estadounidenses o en el desconocimiento de que el cracking también consiste en una técnica de optimización de la programación; son sólo sus aspectos negativos los que se aplican con fines ilegales). El afirmar que todos los crackers son criminales equivale a afirmar que todos los físicos nucleares que sepan cómo crear una bomba atómica son genocidas. La realidad se muestra en conjunto bien distinta. Al igual que hay personas buenas y malas, hay crackers buenos y malos. Mientras que el primero se esfuerza en aprender tanto como pueda, en obtener la mayor experiencia posible, en compartirla, en ayudar a los ingenieros de desarrollo de software indicando dónde residen los puntos débiles de las protecciones y en procurar proteger el software contra el cracking, el último hace exactamente lo contrario: vulnera el software y lo hace circular ilegalmente.

HERRAMIENTAS DE CRACKING ELEMENTALES

Como ya ha quedado dicho, el procedimiento básico consiste en la descompilación de un programa en un lenguaje de programación, a ensamblador en la mayoría de las ocasiones. El cracker podrá estudiar el programa de forma pasiva, desensamblarlo con un *desensamblador* (*descompilador* que efectúa una compilación inversa en ensamblador) para obtener su código ensamblador estático, o bien aplicar un programa de depuración, un *depurador*, para depurar el programa. Al constituir uno de los pasos claves el uso frecuente de algún depurador para realizar el análisis del programa, resulta pertinente intentar evitar su utilización (o al menos, obstaculizarla en gran medida). Razón que conduce a los infractores a aplicar distintos métodos y programas que enmascaren su depurador. El programa más conocido empleado para detectar un depurador y "dejar las cosas en su sitio" se denomina Frogsice.

Si se efectuaran alteraciones al código del programa, éstas deberán guardarse en algún lado. Con este propósito existen varios editores hexadecimales; alternativamente, se puede modificar el código del programa directamente en memoria mediante un *cargador*.

También figuran los *volcadores*, programas diseñados para guardar en disco el contenido de la memoria. Muy útiles para almacenar datos importantes descodificados por el programa automáticamente. ProcDump representa un buen ejemplo: es un volcador y descodificador que también incluye otras muchas funciones.

No se preocupe el lector que no conozca las herramientas y técnicas mencionadas. Se irá familiarizando con ellas progresivamente conforme vaya examinando los ejemplos ofrecidos a lo largo de este libro.

¿MERECE LA PENA PROTEGER EL SOFTWARE O RESULTA INEVITABLE EL CRACKING?

Toda protección está destinada a ser anulada tarde o temprano; ahora bien, no debe ponerse en duda la protección del software, puede apreciarse cómo se extiende por todo el mundo la apatía que encierra esta duda. Los problemas puestos de manifiesto por la piratería de software son de índole muy compleja y exigen soluciones multidisciplinares. La cuestión no radica en cómo impedir la piratería, sino en cuánto se puede aprender acerca del comportamiento de los cracks y en cómo proteger el software durante la mayor cantidad de tiempo posible difundiendo los secretos de los métodos de cracking. Pueden localizarse cientos de libros sobre "hacking" destinados a divulgar los trucos habituales para penetrar en las redes de ordenadores, con el mismo propósito se ha escrito este libro, pero en este caso orientado al software.

CONTENIDOS DEL CD ADJUNTO

El CD adjunto no sólo contiene el código fuente de los ejemplos de este libro, sino algo parecido a lo que podría ser el paquete completo de cracking y anticracking (dentro del conjunto de los programas gratuitos y de código compartido —en inglés, "shareware" —, naturalmente). Se han reunido las últimas versiones de casi todos los compresores y codificadores PE más conocidos, sus correspondientes descompresores y descodificadores, ProcDump incluido, unos cuantos volcadores, generadores de parches y cargadores, editores PE y rastreadores, y calculadoras de ubicaciones, desensambladores, depuradores, Frogslee y muchos otros programas y herramientas —en pocas palabras, la mayoría del software mencionado en este libro y todo lo necesario para empezar a proteger software—. Algunos programas no pudieron incluirse en el CD por rehusar sus autores a dar su consentimiento. Con objeto de compensar esta ausencia, en el capítulo 10 se ofrecerá una lista con las direcciones en Internet donde descargarse el software.

CAPÍTULO 1

MÉTODOS DE PROTECCIÓN Y SUS PUNTOS DÉBILES

En este capítulo se van a describir algunos de los métodos de protección de software más extendidos, señalando sus vulnerabilidades teniendo en cuenta que resulta prácticamente imposible crear una protección de software invencible.

Existen varios métodos fiables de proteger el software sin necesidad de conocer contraseñas, el código fuente, el fichero de registro, o cualquier otro requisito que exija el programa.

CIFRADO

Este método de protección consiste en emplear datos cifrados para descifrarse posteriormente tecleando un número de registro (contraseña, etc.) sobre el que no se realiza ninguna comprobación, tampoco se realiza sobre la clave de descifrado (característica muy importante) —los datos quedan así descifrados—, si se produjera algún error, la clave generada sería errónea.

El único modo de sortear este tipo de protección se reduciría a intentar un ataque masivo, lo que, suponiendo que se haya aplicado un buen algoritmo de cifrado, constituye una tarea casi insuperable. Si el programa emplea un algoritmo tipo RSA y

un código de 2048 bits, quedará protegido con seguridad incluso frente a entidades gubernamentales si bien se ha mejorado mucho el proceso de resolución del problema que supone la factorización (o se halle una solución alternativa).

Desgraciadamente, esta técnica encierra su propia debilidad. Resulta válida sólo hasta que el cracker consiga la clave. Tan pronto la obtenga de un conocido que se haya registrado o la encuentre en Internet, nada evitará que pueda guardar los datos de forma no cifrada.

PROGRAMAS INCOMPLETOS

Otra forma de evitar violar un programa consiste en retirar aquella parte del código que pudiera ser atacado. Resulta bien simple. Al crear un versión para demostración o de código compartido (de funcionalidad limitada) de un programa, resulta preferible empaquetar estas versiones distinguiéndolas de la versión íntegra del programa (y no al revés, donde la versión de demostración se convierte en la definitiva tras introducir el número de serie...). De este modo, se puede suprimir aquella parte del código de la que se pueda prescindir.

Cualquier programa en demostración o de código compartido ("shareware") en el que se evite la presencia material del código constituye un programa incompleto. No hay forma de modificar algo que no existe en el programa. El cracker tendría que programar el código él mismo.

El usuario registrado normalmente recibe la versión íntegra del programa. Lo que, no obstante, exige un mecanismo de copia sofisticado para evitar que la versión completa del programa se difunda ilegalmente. Numerosos programadores han perdido dinero por esta razón.

CLASIFICACIÓN BÁSICA DE LOS TIPOS DE PROTECCIÓN DISPONIBLES

- Duración limitada
- Número limitado de ejecuciones
- Otras restricciones numéricas
- Número de registro o contraseña-número de serie
- Fichero clave

- Programas limitados
- Clave hardware ("dongle")
- Comprobación de la presencia del CD
- Protección contra la copia del CD
- Protección comercial
- Compresores y codificadores para formato PE
- Programas en Visual Basic

Las técnicas de protección tienden a utilizarse de forma conjunta: por ejemplo, se pueden encontrar versiones limitadas por tiempo de un programa que se desbloquea al introducir el número de serie correcto o un fichero temporal, etc.

Para ilustrar lo peligroso que resulta superar las protecciones, mostraremos las vulnerabilidades de algunos de los métodos de protección de software anteriormente mencionados. Ello le podría convencer de la necesidad de empezar a proteger su software.

Duración limitada

Los creadores de software que aplican métodos de protección basados en la duración limitada pretenden asegurarse de que su software no va a volver a funcionar pasado el período de prueba o que sus opciones van a quedar restringidas de una u otra manera.

Figura 1-1. Cuadro de diálogo introductorio de la versión de prueba del programa CloneDVD2

Cualquier programa que utilice este método de protección normalmente guardará la fecha en el momento de su instalación (los algoritmos avanzados lo guardan en varios sitios) —por ejemplo, en registros, ficheros, etc.— y así poder comparar dicho valor con la fecha actual en cada ejecución (varias veces, si fuera posible). Su inconveniente radica en que la fecha actual del sistema depende enteramente del usuario; estos algoritmos se pueden sortear cambiando simplemente la fecha del sistema a otra anterior.

Los últimos mecanismos de protección se basan en métodos mucho más sofisticados para calcular la fecha actual —esto es, a partir de ficheros del sistema—. Su estructura, sin embargo, apenas se ha visto modificada, ni tampoco el procedimiento necesario para su supresión. Puede parecer absurdo que hasta las últimas protecciones comerciales utilicen una estructura similar de protección por duración limitada que sus predecesores —aun cuando se complemente con otro tipo de protección—. La estructura de la mayoría de protecciones de este tipo sigue el modelo siguiente.

```
mov eax, [ebp+123456h]   // obtención de la fecha actual
                         // y almacenándola en el
                         // registro EAX
cmp eax, 1Eh          // comparándola con la fecha/hora
                     //límite, etc. (almacenada en EBX)
jl contiue            // si la fecha actual es menor que
                      // el valor límite, el programa
                      // puede continuar
```

Al cambiar en este caso concreto la última instrucción a una bifurcación incondicional (JMP), el proceso siempre saltará a la etiqueta "continue" independientemente del resultado de la instrucción CMP anterior, así la protección quedará deshabilitada. No resulta necesario enfatizar que sin otro tipo de protección dicho algoritmo no serviría de nada.

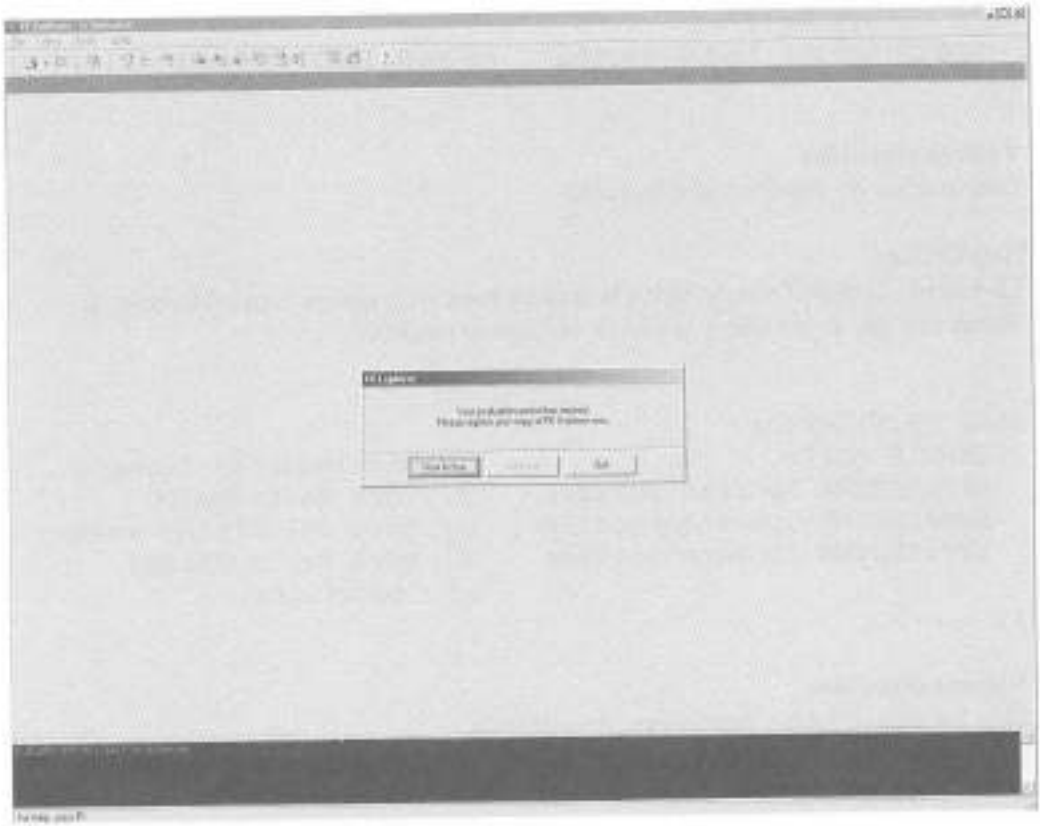

Figura 1-2. Agotada la versión con duración limitada de PE-Explorer

La mayoría de los algoritmos difieren únicamente en la manera de cerciorarse de la fecha actual. A continuación se enumera en una lista las funciones API más utilizadas para este propósito:

GetLocalTime
La función GetLocalTime devuelve la fecha y hora locales.

```
VOID GetLocalTime(
  LPSYSTEMTIME lpSystemTime    // hora del sistema
);
```

Valores obtenidos
Esta función no devuelve ningún valor.

GetSystemTime
La función GetSystemTime almacena la fecha y hora del sistema actuales. La hora del sistema se expresa en UTC ("Universal Time, Coordinated").

```
VOID GetSystemTime(
  LPSYSTEMTIME lpSystemTime    // hora del sistema
);
```

Valores obtenidos
Esta función no devuelve ningún valor.

GetFileTime
La función GetFileTime devuelve la fecha y hora en la que se creó un fichero, la última vez que se accedió y la última vez que se modificó.

```
BOOL GetFileTime(
  HANDLE hFile,                    // manejador de fichero
  LPFILETIME lpCreationTime,       // hora de creación
  LPFILETIME lpLastAccessTime,     // hora del último acceso
  LPFILETIME lpLastWriteTime       // hora de la última
                                   //  escritura
);
```

Valores obtenidos
Si la función devuelve algún valor, no será nulo.
Si la función no devuelve nada, su valor será cero. Para obtener más información sobre el error, invóquese GetLastError.

CompareFileTime
La función CompareFileTime compara dos horas de un fichero.

```
LONG CompareFileTime(
  CONST FILETIME *lpFileTime1,   // primera hora del
                                 // fichero
  CONST FILETIME *lpFileTime2    // segunda hora del
                                 // fichero
);
```

Valores obtenidos
El valor devuelto ha de ser uno de los siguientes:

Valor	Significado
–1	Primera hora del fichero menor que la segunda.
0	Primera hora del fichero igual que la segunda.
1	Primera hora del fichero mayor que la segunda.

GetTickCount
La función GetTickCount devuelve el número de milisegundos que han transcurrido desde que se arrancó el sistema. Depende de la precisión del reloj del sistema. Para obtener la precisión del reloj del sistema utilícese la función GetSystemTimeAdjustment.

```
DWORD GetTickCount(VOID);
```

Valores obtenidos
El valor devuelto será el número de milisegundos transcurridos desde que el sistema se arrancó.

GetTimeZoneInformation
La función GetTimeZoneInformation obtiene los parámetros de la franja horaria actual. Estos parámetros controlan la traducción entre el UTC ("Universal Time, Coordinated") y la hora local.

```
DWORD GetTimeZoneInformation(
    LPTIME_ZONE_INFORMATION lpTimeZoneInformation // franja
                                                  // horaria
);
```

Valores obtenidos
Si se devuelve algún valor, será uno de los siguientes:

Valor	Significado
TIME_ZONE_ID_UNKNOWN	El sistema no puede determinar la franja horaria actual. También se obtiene este error al invocar la función SetTimeZoneInformation con valores de desplazamiento pero sin fechas con que realizar la traducción. Windows NT/2000/XP: se obtiene este valor cuando no se emplea el cambio horario al llegar el verano en la franja horaria actual, ya que no hay fechas de traducción.
TIME_ZONE_ID_STANDARD	El sistema opera en el intervalo indicado por el miembro StandardDate de la estructura TIME_ZONE_INFORMATION. Windows 95/98/Me: se devuelve este valor cuando no se emplea el cambio horario al llegar el verano en la franja horaria actual, ya que no hay fechas de traducción.
TIME_ZONE_ID_DAYLIGHT	El sistema opera en el intervalo indicado por el miembro DaylightDate de la estructura TIME_ZONE_INFORMATION.

Si la función falla, el valor devuelto es TIME_ZONE_ID_INVALID. Si se deseara mayor información sobre el error, invóquese GetLastError.

El cracker alcanzará su objetivo si consiguiera encontrar la función utilizada (normalmente una API) por la protección para cerciorarse de la fecha actual.

Otras restricciones numéricas

La situación con este tipo de protección se asemeja mucho a las basadas en límite temporal. En este caso, el propio programa o algunas de sus características tienen un número de usos limitado (ejecuciones) y no una fecha límite en la que se detenga su funcionamiento. Un programa con este modelo de protección almacena el número de veces que se ha empleado una función relacionada con las restricciones temporales (de nuevo, en varios sitios si fuera posible) y comprueba si todavía se puede utilizar.

Aunque pueda resultar algo más difícil identificar los algoritmos profesionales frente a los programas que utilizan un límite temporal, esta protección será inviable si no se combina con otros mecanismos complementarios. Dado que la mayoría de los programadores no resultan muy inventivos y reinciden en guardar en los mismos sitios el número de veces que se puede utilizar el programa, no resulta nada difícil detectar la protección. Bien es cierto que tampoco son muchos los lugares donde guardar la información: o en el registro de Windows o en ficheros (fácilmente rastreables).

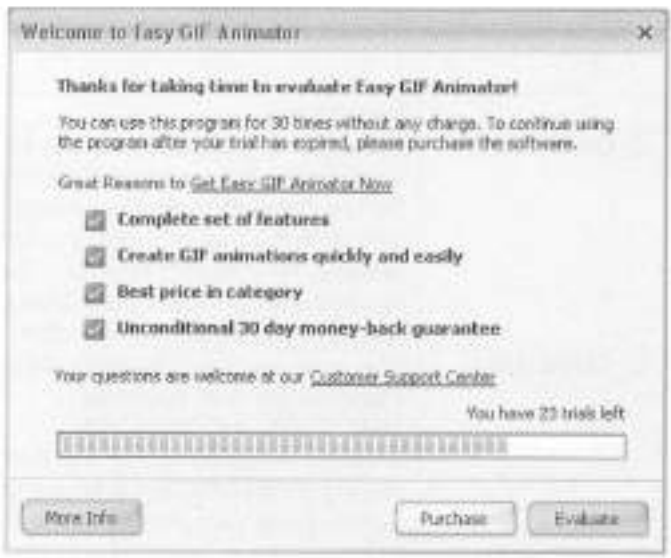

Figura 1-3. Una versión no registrada de Easy GIF Animator permite al usuario ejecutarlo 30 veces únicamente

Número de registro

La protección por número de registro y sus equivalentes (campo de comprobación, combinación secreta de claves, etc.) constituye indudablemente una de las más empleadas en el software actual (empleada principalmente en los programas de código compartido). Con frecuencia esta protección se combina con restricciones de otro tipo que quedan anuladas al introducir el número de registro.

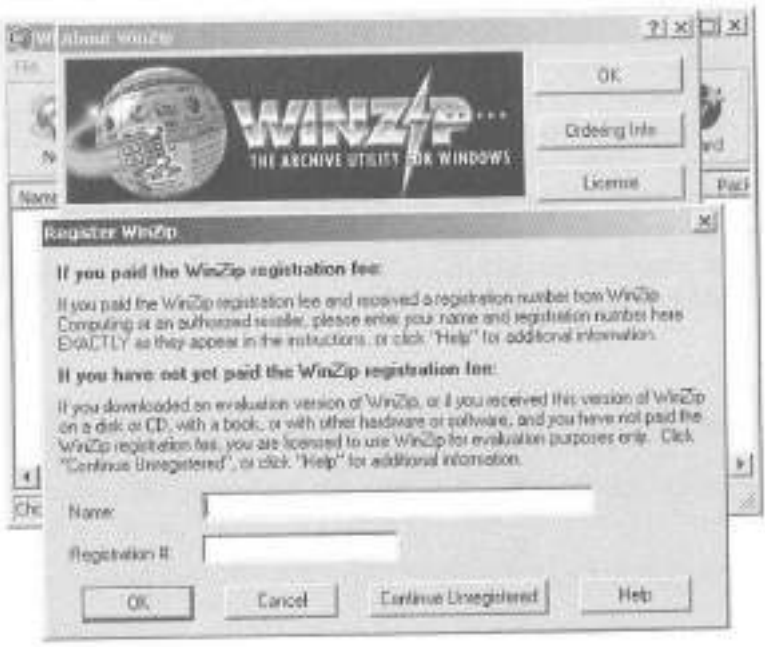

Figura 1-4. El antiguo y buen WinZip con su ventana de registro

Hay muchas formas de llevar a la práctica una protección basada en números de registro. El número se puede almacenar directamente en el código del programa para compararlo con el que se introduzca (éste constituiría el peor de los casos) o bien generarlo dinámicamente, o partir de los datos introducidos o a partir de parámetros concretos del ordenador (configuración de hardware, etc.). Son muchas las alternativas.

El cracker, al intentar suprimir este tipo de protección, intentará obtener el número exigido por el programa depurando el código (véase el capítulo 2), o bien modificará el programa para que parezca que se comporta como si hubiera introducido el número correcto sin hacerlo realmente. En este caso, el cracker generalmente tendrá que modificar el programa en varios lugares dado que la comprobación del número introducido (normalmente guardado en el registro de Windows) con el número real se realiza varias veces —cuando el programa se arranca, cuando se invoca cierta función y, de forma más sofisticada, al azar—.

Normalmente son los principiantes los que adoptan esta estrategia de ataque contra este tipo de protección ya que resulta mucho más simple que intentar entender las rutinas, con frecuencia muy complicadas, utilizadas para la generación de números de registro. Si bien este último caso puede suponer mayor esfuerzo al tener que localizar otros algoritmos de comprobación, el programador podría haber ahorrado trabajo al cracker si hubiera utilizado una sola función para todas las comprobaciones. Éste es uno de los errores que con más frecuencia se cometen. Resulta crítico la manera en la que se haya codificado el programa.

Sin duda alguna, la mejor manera de proteger no sólo un programa sino otros recursos mediante un número de registro, pasa por cumplir con los consejos dados al principio de este capítulo. El número de registro debería emplearse para cifrar aquellas partes del programa que queden accesibles únicamente después de registrarse. Aunque el cracker suprimiera otros mecanismos de protección, no podrían recuperarse las funciones cifradas del programa. Únicamente mediante la introducción de los datos de registro correctos, el programa quedará descifrado y listo para su uso. La seguridad de este tipo de protección resulta directamente proporcional a la seguridad del algoritmo de cifrado.

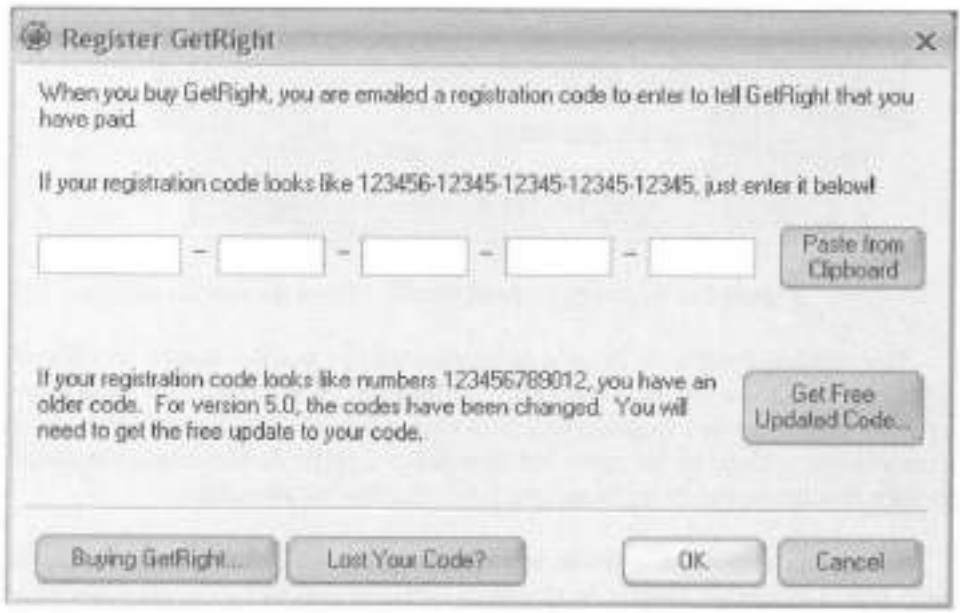

Figura 1-5. Cuadro de diálogo de registro de GetRight

Veamos ahora un sencillo ejemplo de lo que no debería de ser un mecanismo de protección:

```
...
CorrectNumber =
CreateNumber(Keyed_inName,Keyed_inCompany);
if (CorrectNumber == Keyed_inNumber)
{
 MessageBox("Registro correcto",NULL,MB_OK);
 ...
}
else
{
 MessageBox("Registro incorrecto ",NULL,MB_OK);
 ...
}
```

El algoritmo muestra alguno de los errores más comunes cometidos por los autores que aplican este tipo de protección. El primero y más serio, consiste en que el número de registro se devuelva directamente mediante una función (cuyos parámetros están contenidos en la información tecleada por el usuario —su nombre y el de su empresa—). El leerlo tras invocar la instrucción CALL desde un programa de depuración no puede resultar más fácil. El algoritmo de protección nunca debe diseñarse de manera que invoque a una función que a su vez devuelva el número, contraseña, etc., correctos. ¿Por qué no intentar dividir el número real entre los datos de un campo y cifrarlo todo para mayor seguridad? Cuanto más complicado sea el método empleado, más difícil resultará la identificación del número de registro. Otro error consiste en comprobar el número tecleado utilizando bucles con la función "if". No debe de olvidarse lo sencillo que resulta encontrar una instrucción CMP pertinente para visualizar el número de registro o invertir la lógica del algoritmo entero.

Constituye otro error serio, el uso de la evidente función API MessageBoxA y, de hecho, la utilización de una API como tal. El establecer el punto de corte (el mandato dado al programa de depuración para suspender la ejecución del programa y proceder a la depuración —para mayor información véase el final de este capítulo y la sección sobre puntos de corte del capítulo 2—) en esta función API, conducirá directamente al corazón del algoritmo de comprobación. Considérese por otra parte la posibilidad de que el usuario obtenga información a partir de los datos introducidos. ¿Se puede realmente aceptar que nadie conozca las funciones API GetWindowTextA o GetDlgItemTextA?

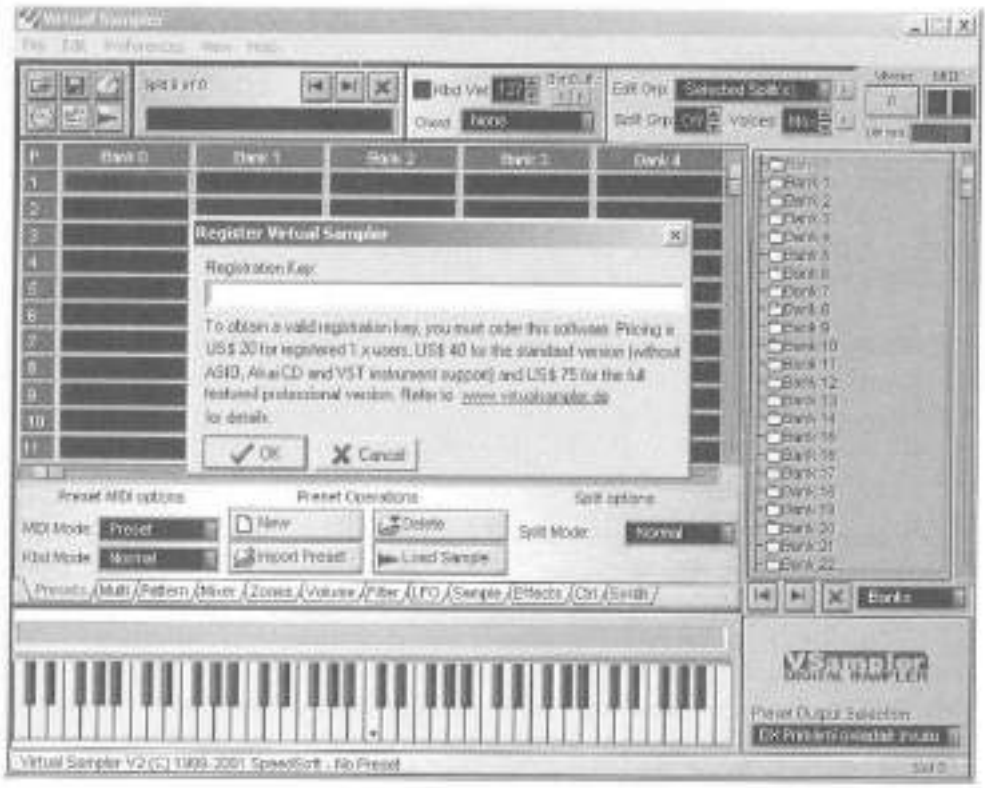

Figura 1-6. Y los números ganadores son ...

Éstas son las robustas características con que los programadores dotan a sus programas para "protegerlos" mediante estos métodos. Al emplear algoritmos similares a éste, las únicas personas contra las que se protege al software son los usuarios legales. Hasta un principiante puede suprimir este tipo de protección. A continuación se enumera una lista de las funciones API más utilizadas con este propósito:

GetDlgItemText / GetDlgItemTextA / GetDlgItemTextW

La funciónGetDlgItemText obtiene el título o texto asociado a un control de un cuadro de diálogo.

```
UINT GetDlgItemText(
  HWND hDlg,      // manejador del cuadro de diálogo
  int nIDDlgItem, // indentificador de control
  LPTSTR lpString, // puntero al buffer de texto
```

```
 int nMaxCount    // tamaño máximo de la cadena
);
```

Valores obtenidos

De ejecutarse con éxito, el valor devuelto por la función indica el número de TCHARs copiados al buffer sin incluir el carácter nulo de finalización.

De ejecutarse sin éxito, el valor devuelto será cero. Si se deseara información más detallada del error, invóquese GetLastError.

GetWindowLong / GetWindowLongA / GetWindowLongW

La función GetWindowLong obtiene información sobre la ventana señalada. También devuelve el valor de 32 bit (largo) en el desplazamiento para la memoria extra de la ventana.

Si se está obteniendo un puntero o un manejador, esta función será sustituida por GetWindowLongPtr. (Punteros y manejadores son de 32 bits en los sistemas Windows de 32 bits y de 64 en los sistemas Windows de 64 bits.) Si se deseara escribir código compatible tanto con las versiones Windows de 32 bits como de 64 bits, utilícese GetWindowLongPtr.

```
LONG GetWindowLongt
 HWND hWnd,  // manejador de ventana
 int nIndex  // desplazamiento del valor que se desea obtener
);
```

Valores obtenidos

De ejecutarse con éxito la función, el valor obtenido será el valor solicitado de 32 bits.

De ejecutarse sin éxito, el valor devuelto será cero. Si se deseara información más detallada del error, invóquese GetLastError.

Si no se hubiera invocado anteriormente SetWindowLong, GetWindowLong devolverá el valor cero para la memoria extra de la ventana o de clase.

GetWindowText / GetWindowTextA / GetWindowTextW

La función GetWindowText copia el texto de la barra de título de la ventana indicada (si existiese) en el buffer. Si la ventana fuera de control, se copiará el texto de control. Sin embargo, GetWindowText no puede obtener el texto de un control en otra aplicación.

```
int GetWindowText(
  HWND hWnd,        // manejador de ventana o control
  LPTSTR lpString,  // buffer de texto
  int nMaxCount     // número máximo de caracteres por copiar
);
```

Valores obtenidos

De ejecutarse con éxito, el valor obtenido será la longitud, en caracteres, del texto copiado excluyendo el carácter nulo final. Si la ventana no tuviera barra de título o texto, si estuviera la barra de título vacía, o si el manejador de ventana o de control fuera incorrecto, el valor obtenido sería cero. Si se deseara información más detallada del error, invóquese GetLastError.

Esta función no puede obtener el texto de un control de edición en otra aplicación.

GetWindowWord

Soportada sólo por compatibilidad con la API de Windows de 16 bits anterior, por otra parte, igual que GetWindowLong.

GetDlgItemInt

La función GetDlgItemInt traduce un control señalado de un cuadro de diálogo en un valor entero.

```
UINT GetDlgItemInt(
  HWND hDlg,          // manejador del cuadro de diálogo
  int nIDDlgItem,     // indentificador de control
  BOOL *lpTranslated, // éxito (estado)
  BOOL bSigned        // valor señalado o no señalado
);
```

Valores obtenidos

De ejecutarse con éxito, la variable indicada por lpTranslated contiene el valor TRUE, y el valor obtenido es la traducción del texto de control.

De no ejecutarse con éxito, la variable indicada por lpTranslated contiene el valor FALSE, y el valor obtenido, cero. Obsérvese que al ser cero un valor traducido posible, no indica por sí sólo error alguno.

Si lpTranslated fuera NULL, la función no indicará información alguna sobre éxito o fracaso.

Si el parámetro bSigned fuera TRUE, indicando que el valor obtenido es un valor entero señalado, asígnese al valor obtenido un tipo entero. Si se deseara información más detallada del error, invóquese GetLastError.

SendDlgItemMessage / SendDlgItemMessageA / SendDlgItemMessageW

La función SendDlgItemMessage envía un mensaje al control indicado en un cuadro de diálogo.

```
LRESULT SendDlgItemMessage(
 HWND hDlg,     // manejador del cuadro de diálogo
 int nIDDlgItem, // identificador de control
 UINT Msg,      // mensaje
 WPARAM wParam,  // primer parámetro del mensaje
 LPARAM lParam   // segundo parámetro del mensaje
);
```

Valores obtenidos

El valor obtenido contiene el resultado tras procesar el mensaje según el mensaje enviado.

REGISTRO INTERACTIVO

Durante los últimos años se viene apreciando una tendencia a realizar de forma interactiva los registros de todo tipo. Esta técnica es aplicable de muchas maneras. Desde el modo más simple de todos, donde el servidor (el de la empresa que realiza los registros) comprueba que la información de registro resulta correcta, al más sofisticado, donde el código del programa nuevo se envía directamente al usuario vía Internet.

Estos métodos representan un notable paso hacia delante en la lucha contra la piratería en el software. La parte más vulnerable de la protección, la comparación efectuada entre la información introducida y la de referencia, se lleva a cabo en el servidor y no en el ordenador del cliente. A pesar de todo, el programa todavía resulta susceptible a muchas formas de ataque. Aunque dependa de los datos que se le envíen, el cracker puede sortear algunas restricciones (esto es, funciones no cifradas). También sigue siendo muy difícil de resolver el problema de las copias de versiones del programa. Una vez que el programa guarda la información sobre su registro ya efectuado, le hace vulnerable.

La mejor forma de protección interactiva consiste en habilitar este tipo de función directamente en Internet en vez de en el código del programa. Los grandes juegos en red o con servidores constituyen un buen ejemplo. Se exige el código de registro (además de las copias legales) para acceder al servidor y jugar al juego interactivamente. Si el código introducido ya lo ha utilizado otro usuario o resulta incorrecto, es evidente que algún usuario no autorizado está intentando acceder al sistema.

Las protecciones interactivas también sirven para que las empresas controlen los registros realizados y vayan creando una base de datos sobre sus usuarios. No obstante, todo ello puede parecer injusto para quienes no tengan acceso a Internet. Si bien estos usuarios pueden tener la voluntad de registrarse, se les fuerza a utilizar versiones piratas para evitar los problemas que conlleva el registro interactivo.

Fichero clave

Debido a que su ejecución resulta mucho más difícil, la protección mediante ficheros clave tiende a evitarse y olvidarse. Sin embargo, el invertir varias horas diseñando un buen algoritmo de protección basado en este método compensará el esfuerzo realizado.

Este método alternativo se asemeja en varias maneras a la protección mediante números de registro. Al igual que en este caso, no se trata de comprobar si es o no correcto el fichero clave, sino de utilizar la información contenida en él —ya sea para descifrar códigos posteriores del programa o incluso para crearlo—. La única gran ventaja del fichero clave consiste en que puede albergar una relativa gran cantidad de datos —desde información de registro hasta el propio código que ha de completar el programa original (como funciones de las que carecía la versión no registrada)—.

Al violar este tipo de protección, el cracker intentará principalmente reconstruir el fichero clave utilizando el código de programa, que a menudo resulta más revelador de su contenido y estructura de lo que sería deseable. Si se asume una buena programación que añada código nuevo al programa basado en esta técnica, volverá imposible la tarea de reconstrucción. O, con mayor exactitud, será posible siempre que el cracker reconstruya el código de programa él mismo. Por tanto, en este caso el cracker habrá de optar por suprimir las restricciones del programa sin el fichero clave. Podrá suprimir otras protecciones (por ejemplo, un límite temporal), pero no será capaz de recuperar las funciones ausentes del programa.

Los desarrolladores normalmente desconocen las posibilidades que brinda este tipo de protección, y así emplean el fichero clave para guardar información sobre el usuario —su nombre, el nombre de su empresa, etc.—. Sin embargo, ésta no resulta una solución nada segura, con esta sencilla estructura, el fichero clave se podrá reconstruir sin grandes problemas. Recuérdese: cuanto más complicada la estructura, mejor.

La protección mediante ficheros clave combinados con el uso de otras técnicas constituye, a mi parecer, uno de los métodos de protección más potentes para proteger el software eficazmente. Al igual que sucede con la protección mediante números de serie, debiera controlarse la difusión de la información sobre registros ya efectuados. Y de nuevo, una alternativa para alcanzar tal finalidad radica en vincular el registro con un ordenador específico (según su hardware, etc.), que para llevarla a cabo, no debe olvidarse que, como sucede con otras protecciones contra el copiado, la necesidad de volver a registrarse al efectuar cambios en el hardware puede desanimar a los posibles usuarios a que compren el software.

Añado a continuación un ejemplo con los errores más frecuentes cometidos al programar una protección basada en ficheros clave. Resultaba obvio que los programadores no son conscientes de las posibilidades que les brinda este tipo de protección.

```
DWORD NOBR;
BYTE Key[9] = {1,2,3,4,5,6,7,8,9}; // campo con los
                // valores correctos del fichero clave
HANDLE File = CreateFile("key_file.key",
GENERIC_READ,FILE_SHARE_READ,NULL,
OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
                // obtención del  manejador de fichero
if (File == INVALID_HANDLE_VALUE)
{
     MessageBox("Error en el registro",NULL,MB_OK);
     ...
return;
}
```

```
BYTE *pMem = new BYTE[GetFileSize(File,NULL)];
                                    // asignación de memoria
ReadFile(File,pMem,
GetFileSize(File,NULL),&NOBR,NULL);
                                // carga del fichero en memoria

for (DWORD i = 0; i < GetFileSize(File,NULL); i++)
{
if (pMem[i] != Key[i]) // ¿contenido del fichero
                                // correcto?

      {
            MessageBox("Registration
            failed",NULL,MB_OK);
            CloseHandle(File);
            ...
            return;
      }
}
MessageBox("Registro realizado",NULL,MB_OK);
...
CloseHandle(File);
delete[] pMem;
```

El mayor error relacionado con este tipo de protección consiste en la comprobación periódica del contenido del fichero comparándolo con el campo que contiene los valores correspondientes del fichero clave, lo que permitiría reconstruir el fichero clave en cuestión de segundos. Como ya se indicó con anterioridad, dichos valores han de utilizarse, y no reducirse simplemente a su comprobación.

A continuación se enumera en una lista las funciones API más utilizadas para este propósito:

CreateFileA / CreateFileW

La función CreateFile crea o abre los objetos siguientes y obtiene un manejador para poder acceder al objeto:

- Consolas

- Recursos de comunicaciones

- Directorios (apertura solo)
- Dispositivos de disco
- Ficheros
- Mailslots
- Pipes

```
HANDLE CreateFile(
  LPCTSTR lpFileName,                    // nombre de fichero
  DWORD dwDesiredAccess,                 // modo de acceso
  DWORD dwShareMode,                     // modo compartido
  LPSECURITY_ATTRIBUTES lpSecurityAttributes, // SD
  DWORD dwCreationDisposition,           // cómo crear
  DWORD dwFlagsAndAttributes,            // atributos del fichero
  HANDLE hTemplateFile                   // manejador de la plantilla
                                         // del fichero
);
```

Valores obtenidos

De ejecutarse con éxito, el valor obtenido será un manejador abierto contra el fichero indicado. Si existiera antes de la invocación a la función y dwCreationDisposition fuera CREATE_ALWAYS o OPEN_ALWAYS, la invocación a GetLastError será ERROR_ALREADY_EXISTS (aun cuando la función termine bien). Si el fichero no existiera antes de la invocación, GetLastError obtendrá el valor cero.

Si la función no se ejecuta con éxito, el valor obtenido será INVALID_HANDLE_VALUE. Si se deseara mayor información sobre el error, invóquese GetLastError.

ReadFile

La función ReadFile lee datos de un fichero, comenzando en la posición indicada por el puntero. Tras una operación de lectura, el puntero se ajusta según el número de bytes realmente leídos a no ser que el manejador de fichero se haya creado con el atributo de solapamiento. Si el manejador de fichero se ha creado para input y output (I/O) coincidentes (solapamiento), la aplicación deberá ajustar la posición del puntero del fichero tras la operación de lectura.

Esta función se ha diseñado tanto para operaciones síncronas como asíncronas. La función ReadFileEx se ha diseñado sólo para operaciones asíncronas. Permite a una aplicación realizar otros procesos durante la operación de lectura de un fichero.

```
BOOL ReadFile(
  HANDLE hFile,                // manejador de fichero
  LPVOID lpBuffer,             // buffer de datos
  DWORD nNumberOfBytesToRead,  // número de bytes en lectura
  LPDWORD lpNumberOfBytesRead, // número de bytes leídos
  LPOVERLAPPED lpOverlapped    // buffer para solapamiento
);
```

Valores obtenidos

La función ReadFile obtiene un valor cuando alguna de las condiciones siguientes resulta verdadera: finaliza una operación de escritura al final del pipe de escritura, se han leído el número de bytes solicitados, o se ha producido un error.

De terminar con éxito, el valor obtenido por la función será distinto de cero.

Si el valor obtenido es distinto de cero y el número de bytes leídos es cero, entonces el puntero del fichero señalará más allá del final del fichero actual en el momento de la operación de lectura. Por el contrario, si el fichero se abrió con FILE_FLAG_OVERLAPPED y lpOverlapped no es NULL, y el valor obtenido es FALSE, GetLastError obtendrá ERROR_HANDLE_EOF cuando el puntero del fichero señale más allá del final del fichero actual.

De no acabar con éxito, el valor obtenido será cero. Si se deseara mayor información sobre el error, invóquese GetLastError.

WriteFile

La función WriteFile escribe datos a un fichero; está diseñada tanto para operaciones síncronas como asíncronas. La función comienza escribiendo datos al fichero en la posición indicada por el puntero del fichero. Tras finalizar la operación de escritura, el puntero se ajusta según el número de bytes realmente escritos a no ser que el manejador de fichero se haya creado con FILE_FLAG_OVERLAPPED. Si el manejador de fichero se ha creado para input y output (I/O) coincidentes (solapamiento), la aplicación deberá ajustar la posición del puntero del fichero tras la operación de escritura.

Esta función se ha diseñado tanto para operaciones síncronas como asíncronas. La función WriteFileEx se ha diseñado sólo para operaciones asíncronas. Permite a una aplicación realizar otros procesos durante la operación de escritura de un fichero.

```
BOOL WriteFile(
  HANDLE hFile,                    // manejador de fichero
  LPCVOID lpBuffer,                // buffer de datos
  DWORD nNumberOfBytesToWrite,     // número de bytes en
                                   //escritura
  LPDWORD lpNumberOfBytesWritten,  // número de bytes escritos
  LPOVERLAPPED lpOverlapped        // buffer para solapamiento
);
```

Valores obtenidos

De acabar con éxito, el valor obtenido por la función será distinto de cero.

De no terminar con éxito, el valor obtenido será cero. Si se deseara mayor información sobre el error, invóquese GetLastError.

SetFilePointer

La función SetFilePointer desplaza el puntero de fichero a un fichero abierto.

Esta función almacena el puntero de fichero en dos valores tipo DWORD. Si se deseara trabajar con mayor comodidad con punteros de fichero mayores a un único valor DWORD, utilícese la función SetFilePointerEx.

```
DWORD SetFilePointer(
  HANDLE hFile,               // manejador de fichero
  LONG lDistanceToMove,       // bytes para desplazar puntero
  PLONG lpDistanceToMoveHigh, // bytes para desplazar puntero
  DWORD dwMoveMethod          // punto de comienzo
);
```

Valores obtenidos

De acabar con éxito la función SetFilePointer y lpDistanceToMoveHigh resulta NULL, el valor obtenido será el DWORD de menor orden del nuevo puntero de fichero. Si lpDistanceToMoveHigh no resulta NULL, la función obtendrá el DWORD de menor orden del nuevo puntero de fichero, y guardará en el DWORD de mayor orden el nuevo puntero de fichero en el LONG señalado por dicho parámetro.

De no acabar con éxito la función y lpDistanceToMoveHigh resulta NULL, el valor obtenido será INVALID_SET_FILE_POINTER. Si se deseara mayor información sobre el error, invóquese GetLastError.

De no terminar con éxito la función y lpDistanceToMoveHigh no es NULL, el valor obtenido será INVALID_SET_FILE_POINTER. Sin embargo, como INVALID_SET_FILE_POINTER es un valor válido para el DWORD de menor orden del nuevo puntero de fichero, deberá invocarse GetLastError para determinar la causa del error. Si se hubiese producido un error, GetLastError obtendrá un valor distinto a NO_ERROR. Consulte al final de este capítulo la sección Observaciones si se desea un ejemplo de este caso.

Si el nuevo puntero de fichero hubiese obtenido un valor negativo, la función fallará, el puntero de fichero no se moverá y el código obtenido por GetLastError será ERROR_NEGATIVE_SEEK.

GetPrivateProfileInt / GetPrivateProfileIntA / GetPrivateProfileIntW

La función GetPrivateProfileInt obtendrá un valor entero asociado a una clave en la sección indicada de un fichero de inicialización.

Esta función se incluye sólo por compatibilidad con las aplicaciones Windows de 16 bits. Las aplicaciones deberían guardar su información sobre inicialización en el registro del sistema.

```
UINT GetPrivateProfileInt(
  LPCTSTR lpAppName,  // nombre de la sección
  LPCTSTR lpKeyName,  // nombre clave
  INT nDefault,       // valor obtenido si no se encuentra el nombre
                      //clave
  LPCTSTR lpFileName  // nombre del fichero de inicialización
);
```

Valores obtenidos

El valor obtenido será el equivalente entero de la cadena que sigue al nombre clave indicado en el fichero de inicialización indicado. Si no se encuentra la clave, el valor obtenido será el valor definido por omisión. Si el valor de la clave es menor a cero, el valor obtenido será cero.

GetPrivateProfileString / GetPrivateProfileStringA / GetPrivateProfileStringW

La función GetPrivateProfileString obtiene una cadena a partir de una sección dada en un fichero de inicialización.

Esta función se incluye sólo por compatibilidad con las aplicaciones Windows de 16 bits. Las aplicaciones deberían guardar su información sobre inicialización en el registro del sistema.

```
DWORD GetPrivateProfileString(
  LPCTSTR lpAppName,        // nombre de la sección
  LPCTSTR lpKeyName,        // nombre clave
  LPCTSTR lpDefault,        // cadena por omisión
  LPTSTR lpReturnedString,  // buffer de destino
  DWORD nSize,              // tamaño del buffer de destino
  LPCTSTR lpFileName        // nombre del fichero de inicialización
);
```

Valores obtenidos

El valor obtenido será el número de caracteres copiados al buffer, excluyendo el carácter nulo de terminación.

Si ni lpAppName ni lpKeyName son NULL y el buffer de destino resulta demasiado pequeño para albergar la cadena solicitada, ésta quedará truncada y seguida por un carácter nulo, el valor obtenido será igual a nSize menos uno.

Si lpAppName o lpKeyName son NULL y el buffer de destino resulta demasiado pequeño para albergar todas las cadenas, la última cadena quedará truncada y seguida por dos caracteres nulos. En este caso, el valor obtenido será igual a nSize menos dos.

WritePrivateProfileString / WritePrivateProfileStringA / WritePrivateProfileStringW

La función WritePrivateProfileString copia una cadena en una sección específica de un fichero de inicialización.

Esta función se incluye sólo por compatibilidad con las aplicaciones Windows de 16 bits. Las aplicaciones deberían guardar su información sobre inicialización en el registro del sistema.

```
BOOL WritePrivateProfileString(
  LPCTSTR lpAppName,  // nombre de sección
  LPCTSTR lpKeyName,  // nombre clave
  LPCTSTR lpString,   // cadena por añadir
  LPCTSTR lpFileName  // fichero de inicialización
);
```

Valores obtenidos

Si la función consigue copiar la cadena al fichero de inicialización, el valor obtenido será distinto a cero.

De no terminar con éxito la función, o limpia la versión guardada con el fichero de inicialización accedido por última vez, el valor obtenido será cero. Si se deseara mayor información sobre el error, invóquese GetLastError.

Programas limitados

Existen muchas maneras de reducir la funcionalidad de los programas. De hecho, ciertas formas de protección pueden considerarse también programas limitados. Sin embargo, aquí me refiero más concretamente a aquellos programas que tienen distintos botones inhabilitados y opciones inaccesibles desde los menús.

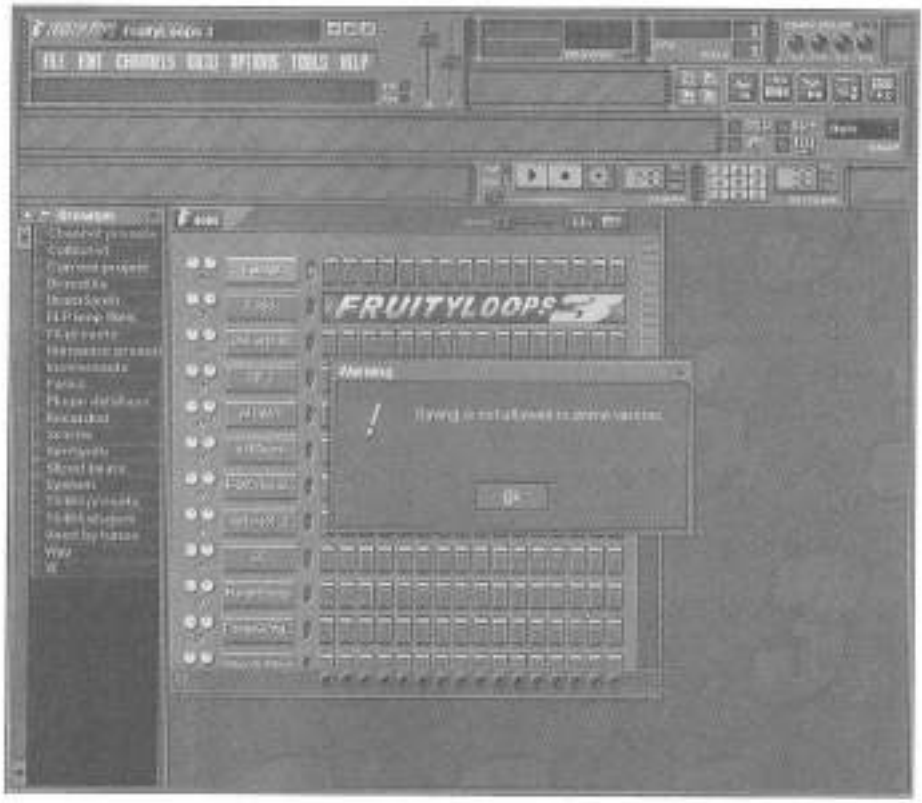

Figura 1-7. Resulta prácticamente imposible guardar nada con esta versión, en demo de Fruity Loops

El problema principal de este tipo de protecciones no radica en la característica de control en sí, sino en el hecho de que dicho mecanismo olvida proteger el propio código del programa. Resulta obvio que de vez en cuando habrá que añadir al programa una o dos de las características deshabilitadas (por ejemplo, para demostrar que la versión en demo no puede realizar lo mismo que la versión completa), de manera que el código del programa controlado por el mecanismo de protección nunca debe permanecer en el programa.

Si la funcionalidad queda habilitada transcurridos unos segundos (lo que indica que el código está presente), resulta recomendable volver a comprobar que se cumple

la condición para su activación o bien aplicar la estrategia recomendada en los tipos de protección anteriormente comentados —cifrar y posteriormente descifrar el código cumplidas las condiciones que permitan su uso, o procesarlos de alguna otra manera, si se dan dichas condiciones—.

El lector podrá comprobar por sí mismo cuán fácil resulta activar un botón o una opción de menú deshabilitados.

Figura 1-8. Inhabilitar la grabación parece ser la manera más frecuente de limitar un programa

Como en otras veces, se enumera en una lista las funciones API más utilizadas para este propósito:

```
InsertMenuItem / InsertMenuItemA / InsertMenuItemW


La función InsertMenuItem inserta una opción de menú nueva en una
posición dada.


BOOL InsertMenuItem(
  HMENU hMenu,          // manejador de menú
  UINT uItem,           // identificador o posición
  BOOL fByPosition,     // significado de uItem
  LPCMENUITEMINFO lpmii // información de la opción de menú
);
```

Valores obtenidos

De terminar con éxito la función, el valor obtenido será distinto de cero.

De no terminar con éxito la función, el valor obtenido será cero. Si se deseara mayor información sobre el error, invóquese GetLastError.

SetMenuItemInfo / SetMenuItemInfoA / SetMenuItemInfoW

La función SetMenuItemInfo modifica la información sobre una opción de menú.

```
BOOL SetMenuItemInfo(
  HMENU hMenu,          // manejador de menú
  UINT uItem,           // identificador o posición
  BOOL fByPosition,     // significado de uItem
  LPMENUITEMINFO lpmii  // información de la opción de menú
);
```

Valores obtenidos

De terminar con éxito la función, el valor obtenido será distinto de cero.

De no terminar con éxito la función, el valor obtenido será cero. Si se deseara mayor información sobre el error, invóquese GetLastError.

EnableMenuItem

La función EnableMenuItem habilita, deshabilita o vuelve gris una opción de menú indicada.

```
BOOL EnableMenuItem(
  HMENU hMenu,        // manejador de menú
  UINT uIDEnableItem, // opción de menú por actualizar
  UINT uEnable        // opciones
);
```

Valores obtenidos

El valor obtenido indica el estado anterior de la opción de menú (MF_DISABLED, MF_ENABLED, o MF_GRAYED). Si la opción de menú no existiera, el valor obtenido sería –1.

EnableWindow

La función EnableWindow habilita o deshabilita la entrada de datos desde el ratón o teclado en la ventana o control indicados. Cuando queda desabilitada, la ventana no recibirá pulsaciones ni del ratón ni del teclado. Cuando se habilite, la ventana sí recibirá dicha información.

```
BOOL EnableWindow(
  HWND hWnd,    // manejador de ventana
  BOOL bEnable  // habilitar o deshabilitar input
);
```

Valores obtenidos

Si la ventana fue deshabilitada anteriormente, el valor obtenido será distinto de cero.

Si la ventana no fue deshabilitada anteriormente, el valor obtenido será cero. Si se deseara mayor información sobre el error, invóquese GetLastError.

Protección Hardware

La protección por hardware ("dongle", en inglés) consiste en una pequeña unidad auxiliar conectada al ordenador, normalmente al puerto paralelo o a algún otro (USB, serie). Esta unidad de hardware resulta muy difícil de duplicar, en ello se basa este tipo de protección. Contiene cierto tipo de memoria con datos y código de programa, lo que le permite comunicarse con el software protegido y auxiliarle en la realización de ciertas tareas: desde comprobar su presencia hasta codificar o descodificar secuencias de caracteres.

Desgraciadamente, a la mayoría de los programadores no se les puede molestar para que estudien la documentación de las funciones ofrecidas por esta unidad de hardware (a diferencia de los crackers, ya que la documentación resulta accesible públicamente) y se ciñen a comprobar si está o no presente. Resulta innecesario señalar que empleando un mensaje de error y una función API del tipo MessageBox, por ejemplo, la protección quedará anulada inmediatamente.

Figura 1-9. Ejemplo de protección por hardware

No obstante, existen otras formas más sofisticadas de protección aplicando las estrategias ya recomendadas al describir los anteriores tipos de protección. La unidad de hardware puede vincularse directamente al programa (el código del programa puede residir en la propia unidad, los valores obtenidos por la unidad se emplearán para descifrar el código pendiente del programa, etc.), lo que significa que la protección no podrá anularse sin la unidad de hardware. Éste parece el mejor modo de proteger el software mediante una unidad de hardware.

Otros tipos de protección más cara demuestran una norma: cuanto más trabaje con acierto el equipo de software en el programa, menos tendrán que preocuparse de proteger su producto ni comprar otro sistema de protección complementario, cuyo manejo desconocen, creyendo que cuanto más caro resulte el precio mejor la protección. Con todo, las empresas gastan una fabulosa cantidad de dinero en sistemas de protección que no son capaces de explotar completamente.

El popular programa 3D Studio Max representa un ejemplo típico. A pesar de su gran popularidad y su alta demanda de copias ilegales (no todo el mundo puede permitirse comprar un programa cuyo precio asciende a varias docenas de euros), los programadores aún no han sido capaces de conseguir una protección eficaz.

Figura 1-10. 3D Studio MAX indica la ausencia de la unidad de hardware

Un ejemplo real de estos programas recibía varios valores desde la unidad de hardware sólo para realizar increíbles operaciones matemáticas, que este autor renunció a comprender por la preocupación que produjo en su salud mental. ¡Los varios centenares de líneas de código tan sólo se remitían a obtener dos valores: 0 si la unidad de hardware no estaba conectada o 1 si sí lo estaba! Este ejemplo ilustra lo pueriles que pueden llegar a ser algunos desarrolladores de software.

Comprobación de la presencia del CD

Comprobar la presencia del CD en la unidad de CD-ROM se ha convertido en un estándar y en el método más empleado, no sólo para proteger software, sino también para comprobar de manera simple si el programa va a ejecutarse con un CD falso en la unidad de disco, lo que podría hacer peligrar su correcto rendimiento.

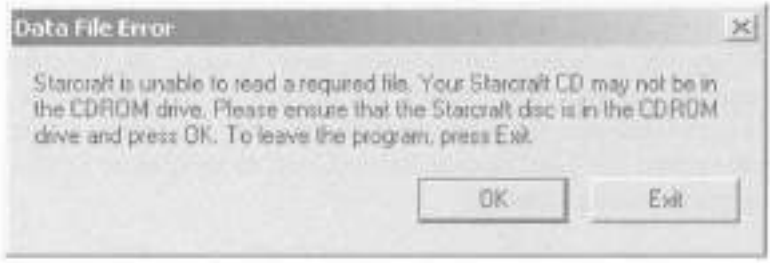

Figura 1-11. Resulta difícil ejecutar StarCraft sin la presencia de su CD

Este tipo de protección se diseñó en el pasado, cuando un grabador de CD era un objeto muy valioso, los CDs no eran tan baratos ni accesibles y no todo el mundo disponía de un CD-ROM. Por lo tanto, el tamaño del programa se reducía eliminando secuencias de

vídeo, música, documentación o código DirectX y se ejecutaba directamente desde el disco duro.

Actualmente existen varias formas de verificar la presencia de un CD concreto en la unidad de CD-ROM: comprobando la etiqueta del CD, comprobando el espacio libre, comprobando los ficheros que se eliminan con frecuencia, etc.

Figura 1-12. Aquí también resulta indispensable el CD

La mayoría de estos algoritmos desgraciadamente se basan en la invocación a una función API bien conocida: GetDriveTypeA, empleada para detectar la unidad CD-ROM en el sistema, lo que facilita su localización y supresión. Además, dichas técnicas resultan absolutamente superfluas cuando prácticamente todo el mundo puede grabar sus CDs. Hoy en día cualquiera puede grabar en CD lo que necesite, este tipo de protección se centra más bien en dificultar la difusión de software ilegal en Internet. El tamaño de una versión íntegra de un juego "limpiado" de esta manera puede quedar reducida a una décima parte de su tamaño original, y transferido cómodamente a través de Internet incluso para aquellos que dispongan de una conexión lenta.

A continuación se enumera en una lista las funciones API más utilizadas para este propósito:

CreateFileA / CreateFileW

- viz. Key-file

GetDiskFreeSpace / GetDiskFreeSpaceA / GetDiskFreeSpaceW

La función GetDiskFreeSpace obtiene información sobre el disco indicado, incluyendo su cantidad de espacio libre.

La función GetDiskFreeSpace no puede abarcar tamaños superiores a 2 GB. Si se desea que la aplicación trabaje con discos de gran capacidad, utilícese la función GetDiskFreeSpaceEx.

```
BOOL GetDiskFreeSpace(
 LPCTSTR lpRootPathName,         // directorio raíz
 LPDWORD lpSectorsPerCluster,    // sectores por cluster
 LPDWORD lpBytesPerSector,       // bytes por sector
 LPDWORD lpNumberOfFreeClusters, // clusters libres
 LPDWORD lpTotalNumberOfClusters // clusters en total
);
```

Valores obtenidos

De terminar correctamente, la función obtiene un valor distinto de cero.

De no terminar correctamente, la función obtiene un valor distinto de cero. Si se deseara mayor información sobre el error, invóquese GetLastError.

GetDriveType / GetDriveTypeA / GetDriveTypeW

La función GetDriveType indica si una unidad de disco se puede retirar, es un disco fijo, un CD-ROM, un disco RAM, o un disco de red.

```
UINT GetDriveType(
  LPCTSTR lpRootPathName   // directorio raiz
);
```

Valores obtenidos

El valor obtenido indica el tipo de disco. Puede ser uno de los valores siguientes:

Valor	Significado
DRIVE_UNKNOWN	No se puede determinar el tipo de unidad.
DRIVE_NO_ROOT_DIR	El directorio raíz es incorrecto, por ejemplo cuando no se ha montado ningún volumen en el directorio.
DRIVE_REMOVABLE	El disco se puede retirar de la unidad.
DRIVE_FIXED	El disco no se puede retirar de la unidad.
DRIVE_REMOTE	Disco remoto (red).
DRIVE_CDROM	CD-ROM.
DRIVE_RAMDISK	Disco RAM (memoria).

GetFullPathNameA / GetFullPathNameW

La función GetFullPathName obtiene todo el directorio ("path") y el nombre de un fichero indicado.

```
DWORD GetFullPathName(
  LPCTSTR lpFileName,  // nombre de fichero
  DWORD nBufferLength, // tamaño del buffer de directorio
  LPTSTR lpBuffer,     // buffer de directorio
  LPTSTR *lpFilePart   // dirección del nombre del fichero en el
                       //directorio
);
```

Valores obtenidos

De terminar la función correctamente, el valor obtenido será la longitud en TCHARs, de la cadena copiada a lpBuffer, sin incluir el carácter nulo de terminación.

Si el buffer lpBuffer resulta demasiado pequeño para contener el directorio completo ("path"), el valor obtenido será el tamaño del buffer, en TCHARs, necesario para guardar el directorio completo ("path"). Por tanto, si el valor resultante fuera mayor que nBufferLength, invóquese la función de nuevo con un buffer suficientemente grande como para albergar el directorio completo ("path").

Si la función no terminase bien por cualquier otra razón, el valor resultante será cero. Si se deseara mayor información sobre el error, invóquese GetLastError.

GetLogicalDrives

La función GetLogicalDrives obtiene una máscara de bits representando las unidades de disco disponibles.

DWORD GetLogicalDrives(VOID);

Valores obtenidos

Cuando la función termina con éxito, el valor obtenido es una máscara de bits con la representación de los discos actualmente disponibles en el sistema. El bit en posición 0 (el menos significativo) indica la unidad A, el de la posición 1, la unidad B, el de la posición 2, la unidad C, y así sucesivamente.

Si la función no terminase bien por cualquier otra razón, el valor resultante será cero. Si se deseara mayor información sobre el error, invóquese GetLastError.

GetLogicalDriveStringsA / GetLogicalDriveStringsW

La función GetLogicalDriveStrings guarda en un buffer una cadena con la que se indica los discos válidos del sistema.

```
DWORD GetLogicalDriveStrings(
  DWORD nBufferLength,  // tamaño de buffer
  LPTSTR lpBuffer       // buffer de la cadena de discos
);
```

Valores obtenidos

De terminar con éxito, el valor obtenido será la longitud, en caracteres, de las cadenas copiadas al buffer, sin incluir el carácter nulo de finalización. Obsérvese que un carácter nulo ANSI-ASCII emplea un byte, pero el carácter nulo Unicode emplea dos.

Si el buffer no es lo suficientemente grande, el valor obtenido será mayor que nBufferLength. Es el tamaño del buffer necesario para albergar la cadena de las unidades de disco.

Si la función no terminase bien por cualquier otra razón, el valor resultante será cero. Si se deseara mayor información sobre el error, invóquese GetLastError.

GetVolumeInformationA / GetVolumeInformationW

La función GetVolumeInformation obtiene información sobre un fichero y volumen del sistema de un directorio raíz indicado.

```
BOOL GetVolumeInformation(
  LPCTSTR lpRootPathName,          // directorio raíz
  LPTSTR lpVolumeNameBuffer,       // buffer del nombre del
                                   //volumen
  DWORD nVolumeNameSize,           // longitud del buffer de
                                   //nombre
  LPDWORD lpVolumeSerialNumber,    // nº de serie del
                                   //volumen
  LPDWORD lpMaximumComponentLength, // longitud máxima
                                    // del nombre del fichero
  LPDWORD lpFileSystemFlags,       // opciones del fichero del
                                   //sistema
  LPTSTR lpFileSystemNameBuffer,   // buffer del nombre del
                                   //fichero del sistema
  DWORD nFileSystemNameSize        // longitud del buffer
                                   //para el nombre del fichero del sistema
);
```

Valores obtenidos

Si se obtuviera toda la información, el valor resultante será distinto de cero.

Si no se obtuviera toda la información, el valor resultante será cero. Si se deseara mayor información sobre el error, invóquese GetLastError.

Compresores y codificadores PE

A pesar de que se dedique en este libro un capítulo íntegro al formato de fichero PE, hay dos términos que conviene aclarar con antelación: codificador PE y compresor PE. La razón es bien simple: puesto que estas dos herramientas representan los métodos de

protección frente al cracking más utilizados y se utilizarán a lo largo de todo el libro, resulta pertinente describirlos con brevedad.

¿Qué es un codificador/compresor PE? Una herramienta que permite codificar (y en el caso de los compresores, comprimir) el código ejecutable del programa. La descodificación se realiza transparentemente en el momento de la ejecución y con ello se evita instalar otro programa. Con este método se dificulta enormemente la edición directa del código y otras técnicas de cracking.

Si se desea más información, acúdase al capítulo 7.

Protección contra la copia del CD

Debido al creciente número de copias de CDs no originales, la mayoría de las empresas especializadas en protección de software se centran en las técnicas anticopia, cuyo fin consiste en prevenir la realización de una copia precisa de un CD. Con posterioridad se tratarán las protecciones comerciales, ahora, los principios básicos de la protección contra las copias de CD.

DETERIORO FÍSICO DEL CD

Los CDs protegidos de este modo deterioran físicamente el CD —por ejemplo, con incisiones, partes ilegibles de la superficie, etc.— con lo que resulta sumamente difícil realizar operaciones de copia con exactitud. Incluso las mejores unidades de CD-ROM no son capaces de crear una copia perfecta de CDs así protegidos, o bien su producción acarrearía docenas de horas con pobres resultados.

A causa del coste que conlleva estas modificaciones especiales y la fabricación de los CDs, este modo de protección apenas se ha empleado en unos pocos casos.

FICHEROS DE TAMAÑO FALSO

Esta técnica bastante extendida consiste en incluir ficheros con un tamaño falso en el CD. Un fichero de este tipo puede alcanzar 1 GB e incluir varios de ellos en un solo CD.

Por esta razón, el tamaño total del CD puede superar aparentemente varios GBs. Lo que impide copiar el CD a un disco y volverlo a grabar en otro CD. La mejor forma de solucionar este problema consiste en duplicar el CD tal cual, ficheros incluidos. La mayoría de estas protecciones o comprueban la presencia de dichos ficheros en el CD, o bien este fichero contiene datos fundamentales para el programa (por ejemplo, el propio programa de instalación). Ésta es la causa de que resulte

esencial copiar los mencionados ficheros. No obstante, existen protecciones oscuras que ni comprueban la presencia de estos ficheros en el CD ni los necesitan para la ejecución del programa. Como ejemplo de este monumental defecto, puede citarse la enciclopedia checa Diderot. Resulta difícil creer que el programa pueda ejecutarse sin problemas tras copiar todo excepto los ficheros de tamaño falso (por no mencionar el hecho de que la protección no realiza ninguna comprobación con el CD).

CDs SOBREDIMENSIONADOS

Este tipo de protección, empleada en el pasado, consistía en producir una densidad de datos superior en el CD original, con lo que se veía aumentada su capacidad. Los CDs así protegidos podrían alcanzar una capacidad ligeramente superior a los 74 minutos, como en esa época aún eran difíciles de conseguir CDs de 80 minutos, de manera que para copiar el CD era preciso omitir algo de información.

En cuanto surgieron los discos de 90 minutos, este método de protección se volvió absolutamente inútil. Incluso desde que existe el modo de escritura "raw", la mayor parte de las grabadoras de CDs actuales son capaces de sobredimensionar los discos (y escribir por encima del límite recomendado) y apurar aún más su capacidad.

TOC ("TABLE OF CONTENTS" EN INGLÉS) ILEGAL

El formato definido por ISO permite únicamente un solo bloque de datos por CD. Cuando un disco se protege de esta manera, contiene varios bloques de datos, lo que lógicamente provoca un error al intentar grabarlo. Existen, no obstante, unos cuantos programas de grabación de CDs que ignoran esta anomalía y funcionan sin inconveniente alguno. Por lo que este método de protección apenas sí supone obstáculo para copiar un CD.

FICHEROS AGRUPADOS

El agrupar los ficheros no impide la copia del CD, pero sí evita suprimir parte de su contenido (vídeo, música, etc.) concentrando todos los datos en uno o dos ficheros bien grandes. Si quisiera eliminar parte del contenido, el cracker deberá repasar la estructura del fichero íntegramente, lo que, debido a su tamaño, supone una tarea de mayor envergadura a que si estuviesen los datos repartidos en varios ficheros pequeños, que suele ser lo más habitual. Esta protección se ha empleado con juegos tales como Quake 3 o StarCraft.

Esta técnica se empleaba más en el pasado, cuando muchos juegos se copiaban directamente al disco duro, cuyo tamaño no era entonces tan grande (como se mencionó anteriormente). Con los bajísimos precios que hoy en día tienen los discos vírgenes y con lo que se venden, sólo unos verdaderos entusiastas protegen los programas de esta manera.

ERRORES FICTICIOS DE SOFTWARE Y OTRAS MANIPULACIONES EN EL PROCESO DE FABRICACIÓN DE LOS CDs

Esta técnica contra la copia de los CDs se emplea actualmente por la inmensa mayoría de las protecciones comerciales. Se basa en la comprobación deliberada (y no física) de errores y otras características identificativas (firmar digitales, etc.) de los CDs, bien difíciles de realizar que, lo más importante, exigen mucho tiempo si se pretende copiar el disco correctamente. Estas características especiales antipirata se utilizan frecuentemente para calcular una clave empleada por un compresor o codificador tipo PE, con el que a menudo se combina este método de protección.

Para producir una copia exacta (clonar), el CD-ROM debe cumplir varias condiciones, entre ellas, la normativa RAW-DAO 96 (que define la capacidad de lectura y escritura de todos los canales de subcódigo). El CD-ROM más extendido con estas características es el TEAC CD-W524. También se pueden encontrar el PRETOR PX-W2410A y el Lite-On LTR-24102B.

La situación difiere mucho entre los distintos tipos de protección. El tema será tratado con mayor detalle en la siguiente sección, donde se estudiarán las protecciones una por una.

Protecciones comerciales

No lleva tiempo decidir qué protecciones deben figurar en esta sección. En teoría, deberían señalarse todas las protecciones desarrolladas con fines comerciales. Por otro lado, no parecería muy sensato incluir aquí todos los programas de protección de código compartido. Razón por la que solamente aparecerán los productos de protección y más extendidos, creados, con algunas excepciones, por grandes empresas y diseñados para proteger el software de manera compleja. Ello excluye algunos codificadores/compresores comunes de tipo PE (véase el capítulo 7), que se tratarán posteriormente.

Resulta estéril describir y analizar todas las protecciones comerciales con detalle. No hay ni una (entre las creadas por ahora) que haya perdurado lo suficiente, y en Internet se pueden encontrar cracks genéricos para la mayoría de ellas. Me remitiré a presentar una breve descripción de los errores más frecuentes que los desarrolladores han cometido en cada uno de los casos.

La lista comienza con las protecciones contra la copia de CDs. Esta lista comprende la mayoría de productos comerciales más y menos conocidos, antiguos y modernos, dedicados a proteger los datos de los CDs (algunos de los cuales también funcionan con DVDs):

Nombre comercial:	Nombre de la empresa:
Alcatraz	KDG
CD-Cops	LINK Data Security Spinner Software
CopyLok	Pan Technology Limited Toolex International N.V.
CDLock	CrypKey
DBB	Effnet
DiscGuard	TTR Technologies Inc.
FADE	Codemasters
LaserLock	MLS LaserLock International
LockBlocks	Dinamic Multimedia
Phenoprotect	Codecult
ProtectCD	VOB
Ring PROTECH	ED-CONTRTIVE
Roxxe	Electronic Publishing Association LLC
SafeDisc	C-Dilla Macrovision Corporation
SafeCast	C-Dilla Macrovision Corporation
SecuROM	Sony
Star Force	Protection Technology Co.
TAGES	Thomson & MPO
The Bongle	Hide & Seek Technologies
The Copy-Protected CD	Hide & Seek Technologies

Puesto que todas estas protecciones se basan prácticamente en el mismo principio y resultan idénticas en muchos aspectos, no sería práctico estudiarlas una a una por separado. Se describirán las más conocidas, en las cuales se han basado las demás.

SAFEDISC

Esta protección antipirata de C-Dilla no es nueva en este mercado. Explota la bien probada combinación de un codificador tipo PE (véase el capítulo 7) y una protección contra la copia creando un vínculo entre ambas. La segunda se basa por un lado, en el gran número de errores practicados en el CD (unos 20.000) y, por otro, en la presencia de una firma digital especial en el disco original, empleada para cifrar y descifrar el programa. Estas propiedades resultan muy robustas y, lo más importante, conlleva una gran cantidad de tiempo copiar un disco con precisión, con lo que realizar copias idénticas un disco se convierte en una tarea extremadamente difícil. Existe un descodificador genérico para la versión anterior de SafeDisc en Internet con el que el programa puede ejecutarse inclusive desde una copia de seguridad sin estas defensas. Además, se ha detectado un error en el algoritmo de cifrado, gracias al cual se puede descifrar un fichero mediante un ataque masivo en un período de tiempo que oscila desde unos segundos a pocos minutos. Así, se

evita la necesidad de disponer del CD original al suprimir la protección. A pesar de los problemas mencionados y de sus constantes actualizaciones, SafeDisc constituye una de las protecciones más extendidas en todo el mundo y la incluyen la mayoría de los juegos. No existe en el mercado ningún producto comercial que pueda competir con ella. Las empresas que la utilizan son conscientes de que no evitarán la difusión ilegal de su software, pero al menos restringirán la copia del CD entre los usuarios comunes.

Figura 1-13. Tecnología de fabricación de CDs protegidos con SafeDisc

SECUROM

En lo que este autor alcanza a conocer, este producto de Sony fue la primera protección contra la copia de CDs que se empleó en grandes cantidades. SecuROM representa una versión "light" de SafeDisc, en la que en verdad esté inspirado. Aunque se haya actualizado constantemente, ha quedado ya obsoleto y no representa gran dificultad anular la protección que brinda.

PROTECTCD

El último producto contra la copia de CDs basado en un codificador PE que se discutirá en esta sección se denomina ProtectCD, de VOB. Es el miembro más reciente de la familia de productos centrados principalmente en el software de juegos. A semejanza de sus hermanos mayores SafeDisc y SecuROM, no aporta ninguna tecnología radicalmente

nueva. Para quien pueda enfrentarse a los dos sistemas anteriores, ProtectCD no constituye ningún problema.

Así llegamos al final de esta breve lista de los productos comerciales mejor conocidos y más extendidos para dotar al software de una protección contra la copia de CDs. Muchos no figuran en esta lista por haber quedado anticuados y resultar muy fáciles de anular (como LaserLock).

Llega ahora el momento de considerar otros productos comerciales de distinta categoría. Se basan en la modificación del software para conseguir su protección (por ejemplo, versiones de duración limitada, protección mediante número de serie, etc.).

ARMADILLO SOFTWARE PROTECTION SYSTEM

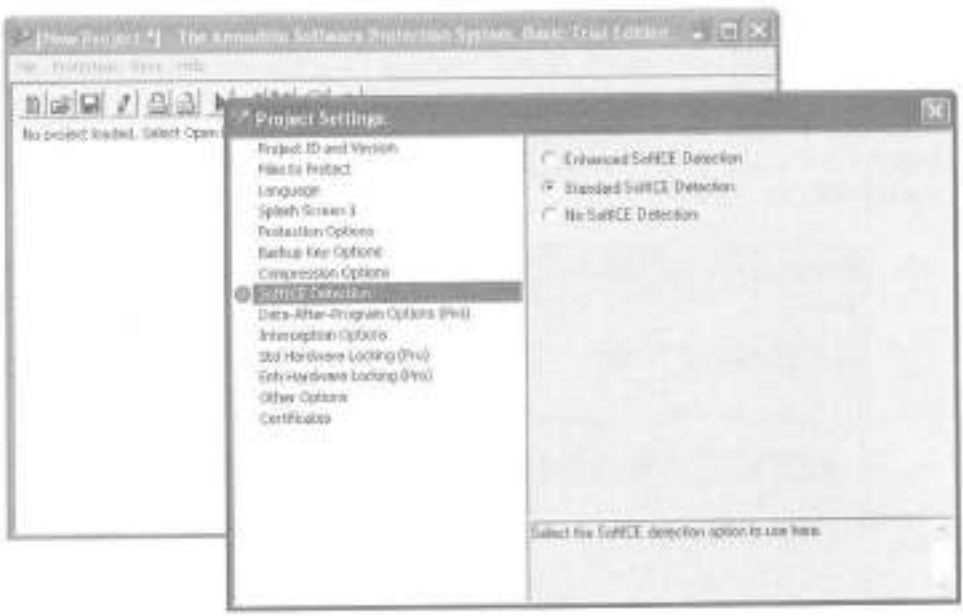

Figura 1-14. Armadillo y sus propiedades

Las versiones anteriores de este producto incurrían en una serie de errores que, dada la protección basada en el cifrado de un fichero PE, podían considerarse muy serios. Exponer los algoritmos de antidepuración ("antidebugging", en inglés), tan pésimamente ocultos que los vuelve detectables casi inmediatamente (para ser más precisos, no se han ocultado en absoluto), convierte al propio bucle de descodificación en el único obstáculo real. Primero descifra el programa y a continuación invoca la función API CreateProcess, que crea un nuevo hilo para el programa protegido. El hilo se crea dejando el parámetro en suspenso (para permitir el descifrado de otras partes del programa), lo que implica que el programa se arranca nada más invocar la función API ResumeThread. Ello facilita muchísimo la tarea al cracker: sólo necesita establecer el punto de corte en esta función y

guardar el contenido del hilo (el programa completamente descifrado) de la memoria al disco. Puesto que no existe ningún otro mecanismo de protección, el fichero no necesita modificarse posteriormente, resulta plenamente funcional y desprotegido —eso, si su estado anterior pudiera considerarse "protegido"—. Puede que un principiante no pueda anular esta protección, pero no es más que un entretenimiento para un experto.

Muchos de los errores se han corregido en las nuevas versiones del producto. Se ha mejorado claramente el bucle de descodificación y los propios algoritmos de antidepuración. Con estas medidas, y aunque continúe resultando fácilmente detectable, aún puede obstaculizar la labor de los crackers, incluso los más expertos. A pesar de todo, existen herramientas de cracking capaces de suprimir automática e íntegramente esta protección en un abrir y cerrar de ojos.

ASPROTECT

Su acertada combinación de un compresor PE y de varias excelentes funciones de protección, sitúan a este producto entre los más destacados en su género. La compresión de ASProtect se basa en un algoritmo PE que constituyó el precedente de este tipo de protecciones y que en su momento se denominó ASPack. Permite alcanzar niveles de compresión muy altos.

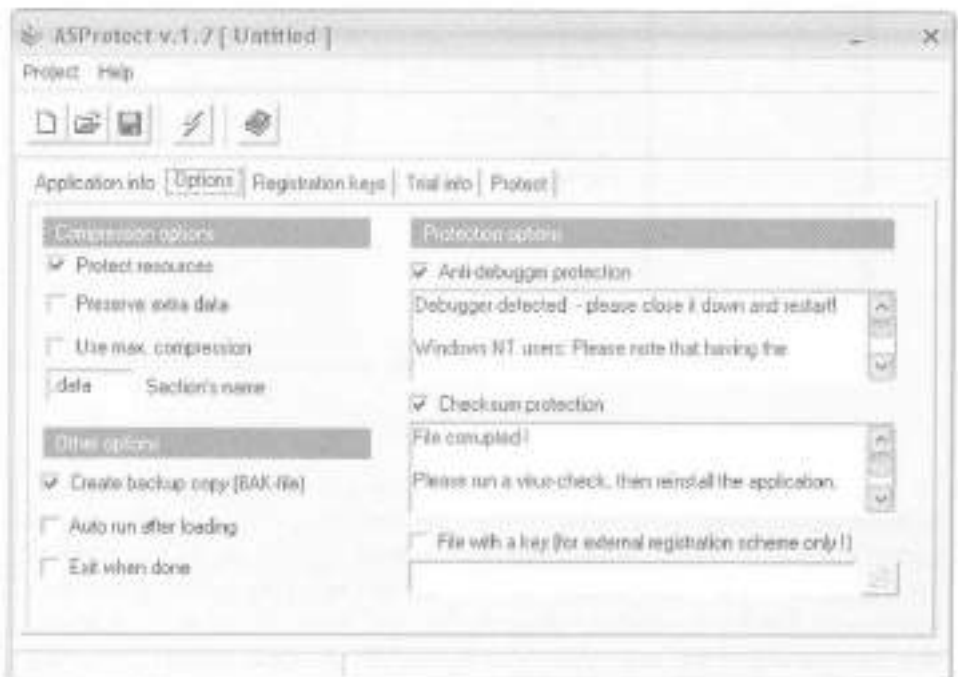

Figura 1-15. ASProtect llama la atención por su buena organización

La protección incluye varias medidas contra el volcado de datos, lo que dificulta enormemente el volcado del programa descifrado, especialmente con ProcDump (programa que se describirá en el capítulo 5). Al poner el autor especial cuidado en el cifrado de la tabla de importaciones original (véase el capítulo 7), el cracker potencial deberá volcar el fichero de cifrado y la tabla de importaciones por separado para posteriormente integrarlas. Puesto que los algoritmos antidepuración no son lo suficientemente sofisticados, el proceso de descompresión manual no supone una gran incomodidad. Si bien algo más difícil de lo normal, a un cracker con veteranía no le debe asustar en absoluto.

La mayor virtud de ASProtect consiste en la protección que confiere a los productos de código compartido cuyas versiones no registradas carecen de ciertas funciones. ASProtect cifra estas funciones según el número de serie (o clave de activación —denomínese como se desee—) de manera que quedan deshabilitadas hasta que el producto protegido se registre. Este método se describió al comienzo del primer capítulo, sabiendo lo que allí se comentó, resulta prácticamente inútil hallar (mediante un depurador) el número de serie exigido. Gracias al algoritmo de cifrado empleado, tampoco valen de nada los ataques masivos; e incluso si el cracker pudiera descomprimir manualmente el fichero, sin conocer la clave de registro no le valdría de nada.

SALESAGENT

El lector puede haberse encontrado con esta protección sin saberlo. Muy típica de los productos en demo, donde se limita el uso del programa protegido a cierto período de tiempo. También denominados en inglés "try and buy", puesto que al arrancar el programa protegido se presenta un cuadro de diálogo (que, sin embargo, se denomina de diversas maneras) con la posibilidad de comprar el producto a través de Internet y de probarlo (suponiendo que no se haya consumido ya el período de prueba).

Esta protección se ha utilizado, por poner algunos ejemplos, en los productos de Symantec (Norton Utilities, Norton Antivirus) y en prácticamente todos los de Macromedia. El código de programa no se cifra en modo alguno, no se valida la integridad de la memoria, sólo la integridad de los datos de los ficheros. Todo lo que el cracker ha de hacer es crear un cargador (véase el capítulo 6) o deshabilitar la propia rutina de comprobación, cuya identificación es cuestión de unos pocos minutos.

VBOX

VBOX parece ser el programa de protección más antiguo, junto con SalesAgent, basado en la duración limitada de los productos. VBox utiliza el cifrado de ficheros, cierto número de comprobaciones de integridad, y resulta bastante sofisticado en términos generales. Desgraciadamente, los desarrolladores no consideraron la peor arma contra el descifrado de ficheros, esto es, el volcado de datos. Puesto que un fichero descifrado puede guardarse con facilidad de la memoria a un disco, esta protección resulta muy fácil de

superar. Se ha venido actualizando continuamente, lo que puede comprobarse a partir del gran número de versiones (las primeras solían denominarse Timelock), y constituye una solución excelente para la versiones de los programas en demo.

Programas en Visual Basic

Uno de los mayores inconvenientes de los programas de Visual Basic radica en que el autor a duras penas controla el resultado final del código (tras la compilación). Ello se debe a que en su mayor parte hace referencias y llamadas a DLL de Visual Basic, generalmente a las de tipo msbvmXX.dll (donde XX indica la versión del lenguaje).

Los crackers no son muy aficionados a estos programas ya que resulta difícil orientarse y el código resulta confuso; lo que le obliga a invertir la mayor parte del tiempo en el código de la DLL. No obstante, la tarea se simplifica muchísimo empleando programas de depuración para Visual Basic (como SmartCheck o VBDebugger), lo que reduciría a pocos minutos la tarea de anular la mayoría de las protecciones.

No es en absoluto cierto que bloquear las protecciones de estos programas sea necesariamente más fácil que con otros lenguajes de programación. Que el código del algoritmo de protección en sí forme parte de alguna clase de librería que resulte inaccesible al programador supone una desventaja en buena parte de las ocasiones, pero no siempre sucede así.

Anular estos programas con frecuencia exige encontrar la función específica invocada por el programa de la librería. Bien se pudiera pensar que esto constituye una ventaja, puesto que estas funciones no están bien documentadas como las API, por ejemplo.

Aunque pudiera ser cierto, los crackers no pierden el tiempo y compilan manuales de referencia para las funciones de todo tipo. Por esta razón, ya se conoce en un buen número de funciones de muy diversa índole. Como entre las más destacadas figuran las funciones que comparan números o cadenas de caracteres (números de serie, contraseñas, etc.), el cracker puede calcular los números necesarios en cuestión de segundos sin tener que analizar el código de programa. No tiene más que establecer el punto de corte en la función pertinente y leer el número requerido.

En la práctica, el cracker contemplará todas las funciones que se hayan podido emplear y elegirá las más probables, el éxito estaría casi asegurado.

También juega un papel importante la versión del lenguaje. Al aumentar el número de funciones con las versiones superiores, las posibilidades del programador

también son mayores (como podrá tal vez elegir una función poco conocida), y así disminuyen las facilidades del cracker.

Visual Basic no resulta idóneo para desarrollar aplicaciones "seguras", debiera descartarse para codificar software de protección de calidad. Puesto que hay donde elegir, se puede escoger cualquier otro lenguaje —Ensamblador si fuera posible—. Si se optase por Visual Basic como herramienta de programación, deberá renunciarse a un buen mecanismo de protección. El lenguaje no fue diseñado con ese objetivo. En todo caso, la decisión depende del programador.

Para ilustrar esta afirmación, se reseñan a continuación los tres métodos más conocidos para comparar dos valores en Visual Basic. Resultan muy útiles para calcular números de serie y contraseñas de todo tipo.

COMPARACIÓN DE CADENAS DE CARACTERES

Este método constituye la forma más sencilla de comparación. Resulta facilísimo hallar el número buscado.

```
If "Correct_password" = "Keyed_in_password" then
GoTo registration_succeeded
Else
GoTo error
End if
```

Posibles puntos de corte

__vbastrcomp or __vbastrcmp (STRing COMpare)

Búsquense combinaciones concretas de los siguientes bytes: 56,57,8b,7c,24,10,8b,74,24,0c,8b,4c,24,14,33,c0,f3,66,a7 (cuando se definan puntos de corte con VB6, el nombre de la función debe procederse con msvbvm60!, por ejemplo, bpx msvbvm60!__vbastrcomp).

COMPARACIÓN VARIABLE (TIPO DE DATOS VARIABLE)

La ventaja de este método de comparación reside en evitar el uso de las funciones conocidas anteriormente mencionadas.

```
Dim Correct As Variant, keyed_in As Variant
Correct = Correct_password
keyed_in = Text1.Text
If Correct = keyed_in Then
GoTo registration_succeeded
Else
GoTo error
End If
```

Posibles puntos de corte

__vbavartsteq (VARiant TeST EQual)

COMPARACIÓN VARIABLE (TIPO DE DATOS LARGO)

Este método resulta casi idéntico al anterior, la única diferencia radica en el tipo de datos utilizados. Su ventaja reside en que la rutina de comparación no está incluida en ninguna librería, sino en el propio programa y por lo tanto no se puede utilizar ningún punto de corte específico. Por el tipo de datos utilizado, la contraseña sólo puede consistir en números.

```
Dim Correct As Long, keyed_in As Long
Correct = 12345
keyed_in = Text1.Text
If Correct = keyed_in Then
GoTo registration_succeeded
Else
GoTo error
End If
```

En vez del tipo de datos 'Long', se puede utilizar 'Single', 'Double', 'Integer', 'Byte' o 'Currency'.

Aún quedan más funciones.

CONVERSIÓN DEL TIPO DE DATOS

A continuación se indican algunas de las funciones utilizadas para convertir datos de distinto tipo:

```
String to Byte or Integer: __vbai2str
String to Long: __vbai4str
String to Single: __vbar4str
String to Double: __vbar8str
String to Currency: VarCyFromStr
Integer to String: VarBstrFromI2
```

TRANSFERENCIA DE DATOS

La transferencia de datos en memoria se realiza mediante una de las siguientes funciones:

```
String to memory: __vbaStrCopy
Variant to memory: __vbaVarCopy nebo __vbaVarMove
```

OPERACIONES MATEMÁTICAS

Existe cierto número de operaciones matemáticas empleadas para calcular contraseñas y números de serie. A continuación se enumera una lista con las funciones más utilizadas con este fin.

Suma: `__vbavaradd`
Sustracción: `__vbavarsub`
Multiplicación: `__vbavarmul`
División: `__vbavaridiv`
XOR: `__vbavarxor`

MISCELÁNEA

Esta lista finalizará señalando otras funciones útiles:

`__vbavarfornext` – con bucles de 'For' y 'Next'
`__vbastrvarval` – para obtener un valor en cierta posición de la cadena
rtcMsgBox – MessageBox, muy útil.

Otras vulnerabilidades de las protecciones actuales

El primer paso de un cracker que intente anular la protección de un programa concreto consiste en su identificación. Resulta inviable con ficheros voluminosos examinar una por una las líneas del código del programa con objeto de buscar el algoritmo de protección. Razón por la que algunos crackers han diseñado distintos métodos para

localizarlos e identificarlos con exactitud tan rápidamente como sea posible. Estos métodos abundan, los más eficaces se describirán en los capítulos 2 y 3. Suelen utilizar servicios de desensamblaje y depuración. Lo que explica las referencias a las funciones API más frecuentemente utilizadas en las protecciones mencionadas anteriormente. A continuación se podrá comprobar lo fácil que resulta identificar la protección sabiendo las funciones (entre otras cosas) empleadas en ella. Nunca se insistirá lo suficiente para evitar emplear funciones API al codificar la protección de un programa. Estas funciones constituyen el punto de entrada más frecuente para que el cracker penetre en el programa y le conduzca al mecanismo de protección. El programador deberá responder a esta crucial cuestión al desarrollar una protección: ¿cómo se procederá cuando se cree un número de registro incorrecto, cuando haya transcurrido la duración de un programa en prueba o cuando se detecte un intento de anular el programa de protección, etc.?

Apenas unas pocas personas afrontan este problema crucial a pesar de que constituye la piedra angular de la protección del software, y que por tanto, puede causar efectos en la seguridad inimaginables. Existen actualmente protecciones cuyos desarrolladores no se han privado de utilizar funciones API bien conocidas, como MessageBoxA, que informa a los usuarios, por ejemplo, sobre si el número introducido es o no correcto; todo ello hace inútil la protección. Hay programadores que ignoran que la protección debe abarcar el modo en el que se informa al usuario. Obviamente la ventana que informa al usuario debe ubicarse en algún sitio e invocarse desde algún otro. No hay mejor lugar que en las inmediaciones del algoritmo crucial para su existencia, esto es, el algoritmo de protección.

La mayor parte de los programas de código compartido recuerdan de varias maneras al usuario la necesidad de registrar el programa. Quizá la más frecuente sean los cuadros de advertencia —las denominadas, en inglés, pantallas NAG ('to nag': fastidiar, molestar)—. Todos conocemos las inoportunas ventanas que nos recuerdan que estamos utilizando una versión no registrada del programa. Su papel en la protección no es baladí. Consiste en molestar al usuario hasta el punto que registre o compre el producto original.

La situación es idéntica a la descrita en la sección anterior. Se utilizan funciones API muy conocidas para la creación de ventanas; incluso cuando los programadores sean cuidadosos, aún hay otros métodos que pueden aplicarse a las ventanas (véase el capítulo 2). No se debe crear ninguna ventana, diálogo, ni nada similar que no sea absolutamente necesario. ¿Acaso no basta con informar al usuario sobre el registro del programa en la ventana "Acerca de ..."?

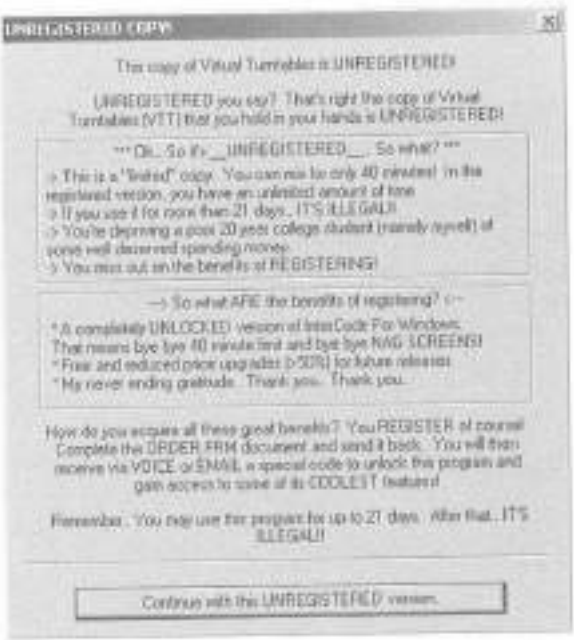

Figura 1-16. Cuadro de diálogo con el que se promete al usuario muchas más funcionalidades si llega a registrarse

Una última recomendación: una de las características más importantes del algoritmo de protección, a menudo infravalorada, consiste en la respuesta que dé a diversas situaciones. Téngase siempre en cuenta la existencia de algún otro método difícil de identificar que pudiera utilizarse para informar al usuario en vez de un cuadro de diálogo o una ventana. Más cierto aún si cabe referido a las partes críticas de un algoritmo de protección. Varias protecciones actuales reaccionan deteniendo el programa o bloqueándose, por ejemplo, al detectar un programa de depuración en funcionamiento. Si bien no constituye el mejor método en lo que al usuario concierne, puesto que ignora por qué el programa se bloquea, sí representa probablemente el mejor método en lo que respecta a seguimiento y seguridad. Una cosa es el seguimiento de la actividad para vulnerar un programa y otra bien distinta, la respuesta a dicha actividad. Este libro dará cuenta de muchos algoritmos de detección junto a otras medidas de protección (algunas veces conjuntamente).

Otro problema de seguridad hallado en varias protecciones actuales consiste en un uso descuidado de las cadenas de caracteres. Como se verá en el tercer capítulo dedicado al desensamblaje, resulta muy sencillo encontrar la ubicación del algoritmo de protección interrogando a las cadenas empleadas por el programa. Considérese por un momento si resulta absolutamente necesario que el programa guarde las cadenas "registro satisfactorio" o "número de serie incorrecto". Este problema se puede acometer de diversas maneras según se describirá en el capítulo tercero.

Antes, he aquí una lista con las funciones API relativas a ventanas:

BitBlt

La función BitBlt realiza una transferencia en bloque de los bits de color correspondientes a un rectángulo de píxeles desde el dispositivo contextual (DC) origen indicado a otro de destino.

BOOL BitBlt(

```
HDC hdcDest, // manejador al DC de destino
int nXDest,  // coordinadas X del ángulo superior izquierdo de
             //destino
int nYDest,  // coordinadas Y del ángulo superior izquierdo de
             //destino
int nWidth,  // anchura del rectángulo de destino
int nHeight, // altura del rectángulo de destino
HDC hdcSrc,  // manejador al DC
int nXSrc,   // coordinadas X del ángulo superior izquierdo de
             //origen
int nYSrc,   //// coordinadas Y del ángulo superior izquierdo de
             //origen
DWORD dwRop  // código de operación raster
);
```

Valores obtenidos

De tener éxito, la función obtiene un valor distinto de cero.

De no tener éxito, la función obtiene un valor igual a cero.

Windows NT/2000/XP: Si se deseara mayor información sobre el error, invóquese GetLastError.

CreateWindow

La función CreateWindow crea una ventana solapada, emergente o esclava ('overlapped', 'pop-up', o 'child'). Determina la clase, título, estilo, y (opcionalmente) la posición inicial y tamaño de la ventana. La función también define, si existiera, la ventana maestra o propietaria, y el menú de la ventana.

Si se desearan utilizar estilos de ventana complementarios distintos a éstos, utilícese la función CreateWindowEx.

```
HWND CreateWindow(
  LPCTSTR lpClassName,  // nombre de clase
  LPCTSTR lpWindowName, // nombre de ventana
  DWORD dwStyle,        // estilo de ventana
  int x,                // posición horizontal de la ventana
  int y,                // posición vertical de la ventana
  int nWidth,           // anchura de la ventana
  int nHeight,          // altura de la ventana
  HWND hWndParent,      // manejador de la ventana maestra
```

```
HMENU hMenu,        // manejador de menu o identificador de la
                    //ventana esclava
HINSTANCE hInstance, // manejador a la ocurrencia de la
                     // aplicación
LPVOID lpParam      // datos de creación de la ventana
);
```

Valores obtenidos

De tener éxito, la función obtiene un manejador de ventana nueva.

De no tener éxito, la función obtiene valor NULL. Si se deseara mayor información sobre el error, invóquese GetLastError.

Cuando esta función falla, lo suele hacer por las siguientes razones:

- un valor de parámetro incorrecto
- la clase del sistema la registró un módulo distinto
- está instalado el enlace WH_CBT y devuelve un código de error
- el procedimiento de ventana falla por WM_CREATE o WM_NCCREATE

CreateWindowEx / CreateWindowExA / CreateWindowExW

La función CreateWindowEx crea una ventana solapada, emergente o esclava ('overlapped', 'pop-up', o 'child') de modo extendido; por otra parte, esta función resulta idéntica a CreateWindow. Si se deseara

mayor información sobre cómo crear una ventana y una descripción completa de otros parámetros para CreateWindowEx, véase CreateWindow.

```
HWND CreateWindowEx(
 DWORD dwExStyle,      // estilo extendido de ventana
 LPCTSTR lpClassName,  // nombre de clase
 LPCTSTR lpWindowName, // nombre de ventana
 DWORD dwStyle,        // estilo de ventana
 int x,                // posición horizontal de la ventana
 int y,                // posición vertical de la ventana
 int nWidth,           // anchura de la ventana
 int nHeight,          // altura de la ventana
 HWND hWndParent,      // manejador de la ventana maestra
 HMENU hMenu,          // manejador de menú o identificador de la
                       //ventana esclava
 HINSTANCE hInstance,  // manejador a la ocurrencia de la
                       //aplicación
 LPVOID lpParam        // datos de creación de la ventana
);
```

Valores obtenidos

De tener éxito, la función obtiene un manejador de ventana nueva.

De no tener éxito, la función obtiene valor NULL. Si se deseara mayor información sobre el error, invóquese GetLastError.

Cuando esta función falla, lo suele hacer por las siguientes razones:

- un valor de parámetro incorrecto

- la clase del sistema la registró un módulo distinto

- está instalado el enlace WH_CBT y devuelve un código de error

- el procedimiento de ventana falla por WM_CREATE o WM_NCCREATE

SendMessage / SendMessageA / SendMessageW

La función SendMessage envía el mensaje indicado a una o varias ventanas. Invoca al procedimiento de ventana según la ventana indicada y no devuelve el control hasta que el procedimiento de ventana haya procesado el mensaje.

Si se deseara enviar un mensaje y obtener el control inmediatamente, utilícense las funciones SendMessageCallback o SendNotifyMessage. Para enviar un mensaje a un hilo con una cola de mensajes y obtener el control inmediatamente, véanse las funciones PostMessage o PostThreadMessage.

LRESULT SendMessage(

HWND hWnd, // manejador de la ventana de destino

UINT Msg, // mensaje

WPARAM wParam, // primer parámetro del mensaje

```
  LPARAM lParam   // segundo parámetro del mensaje
);
```

Valores obtenidos

El valor obtenido indica el resultado del procesamiento del mensaje; dependerá del mensaje enviado.

ShowWindow

La función ShowWindow define el estado de visibilidad de la ventana indicada.

```
BOOL ShowWindow(
  HWND hWnd,     // manejador de ventana
  int nCmdShow   // estado de visibilidad
);
```

Valores obtenidos

Si la ventana ya resultaba visible anteriormente, el valor obtenido será distinto de cero.

Si la ventana estaba oculta anteriormente, el valor obtenido será igual a cero.

UpdateWindow

La función UpdateWindow actualiza el área cliente de la ventana indicada enviando un mensaje WM_PAINT a la ventana si su región de actualización no estuviera vacía. La función envía un mensaje WM_PAINT directamente al procedimiento de ventana de la ventana indicada saltándose la cola de aplicación. Si la región de actualización estuviera vacía, no se enviaría ningún mensaje.

```
BOOL UpdateWindow(
  HWND hWnd   // manejador de ventana
);
```

Valores obtenidos

De tener éxito, la función obtiene un valor distinto de cero.

De no tener éxito, la función obtiene un valor igual a cero.

Windows NT/2000/XP: si se deseara mayor información sobre el error, invóquese GetLastError.

Por último, otra función imprescindible:

HMEMCPY

Esta función API utiliza memoria (RAM) para manejar los datos, normalmente caracteres en una cadena. Por esta razón resulta muy útil para penetrar en programas que emplean protecciones basadas en cadenas de caracteres —números de serie, contraseñas, etc.—. No se suele utilizar para penetrar en programas como Delphi o VisualBasic, que no trabajan con las funciones API estándar GetWindowTextA ni GetDlgItemTextA para leer el texto definido por el usuario. Se podrá estudiar con más detalle esta función en el capítulo 9, donde se incluyen ejemplos prácticos.

CONCLUSIÓN

En este capítulo se han repasado tanto las técnicas actuales de protección de software como sus puntos débiles. A su luz, los desarrolladores podrán plantearse qué tipo de errores han cometido y qué soluciones aplicar. Precisamente éste es el objetivo de los capítulos siguientes.

CAPÍTULO 2

PROTECCIÓN CONTRA LOS PROGRAMAS DE DEPURACIÓN

Resulta prácticamente imprescindible para cualquier programador actual disponer de un buen depurador. Le permitirá encontrar prácticamente cualquier detalle de un programa en cuestión de segundos para poder examinarlo y modificarlo, así como buscar posibles errores. Desgraciadamente, un depurador puede utilizarse no sólo para optimizar el propio código, sino el software de otros autores. Auxiliado con un buen conocimiento de ensamblador, un depurador normal puede convertirse en un arma peligrosa. No es preciso recordar a quienes conozcan el potencial real de los mejores depuradores que constituye un tipo de software vital y de la mayor importancia para el cracker. El lector lo podrá comprobar a lo largo de este libro por sí mismo. Merece la pena, por tanto, intentar obstaculizar la utilización de depuradores.

Nunca deberán considerarse los algoritmos antidepuración el fundamento de la protección de un programa. Muchos expertos afirman con razón que en vez de crear programas antidepuradores, los desarrolladores deberían invertir más tiempo diseñando el sistema de protección adecuado. El utilizar algoritmos antidepuración puede ser una buena idea, pero deben jugar un papel complementario en la protección.

En este libro no se recoge una lista completa de todos los algoritmos antidepuradores que, además, se pueden encontrar en Internet. Y no porque haya demasiados, sino porque algunos se han quedado anticuados y resultan inútiles —la

mayoría sencillamente no funcionan—. Por eso, la lista aquí presentada contiene los algoritmos que aún se pueden utilizar con distinta suerte o bien porque debido a su fama resultaba obligatorio reseñar.

DEPURADORES MÁS HABITUALES

SoftICE

El número de depuradores actualmente disponible es extraordinario, pero ni uno solo de ellos puede utilizarse para el cracking como éste. Habrá muchos donde elegir, pero rey sólo hay uno. No resulta exagerado afirmar que lo utilizan el 99,9% de los crackers de todo el mundo. Sus muchos años de presencia en el mercado internacional del software le han deparado una posición imbatible, que le convierten prácticamente en la única opción para quienes quieran realmente programar con seriedad. Gracias a estas propiedades y virtudes, este software goza de gran popularidad especialmente entre los crackers. Nadie que tenga siquiera un interés secundario en el cracking podrá desenvolverse sin esta herramienta.

TRW constituye otro depurador de gran calidad. Aunque sus autores insistan en que supera a SoftICE en muchos aspectos, la popularidad de este programa sigue sin superarse.

USO ELEMENTAL DE SOFTICE

Ser capaz de trabajar con este soberbio programa forma parte de las aptitudes esenciales no sólo de cualquier cracker, sino de todo buen programador. Veamos al menos los mandatos fundamentales de SoftICE; el resto podrá encontrarse en la sección de referencia del libro y en la magnífica ayuda del programa.

Configuración del programa

Antes de utilizarse por vez primera, SoftICE debe quedar configurado correctamente. El paso más importante en el proceso de configuración consiste en cargar las funciones exportadas por las DLLs pertinentes para que los puntos de corte puedan trabajar con estas funciones. En nuestro caso concreto, los puntos de corte van a apuntar la mayoría de las veces a funciones API, lo que nos obliga a cargar sus dos DLLs más importantes: kernel32.dll y user32.dll.

Únicamente los usuarios de Windows 9x/Me precisan realizar esta operación puesto que la versión de SoftICE para Windows NT/2000/XP carga las dos librerías automáticamente. El programa 'Symbol Loader' (en castellano, cargador de símbolos, incluido en la instalación de SoftICE, puede cargar también ambas librerías, pero resulta mucho más rápida la alternativa manual: edítese el fichero winice.dat ubicado en el

directorio de instalación de SoftICE. Se puede abrir con notepad; búsquese la sección siguiente:

```
;*****Examples of export symbols that can be included for
Windows 95 *****;
Change the path to the appropriate drive and directory
;EXP=c:\windows\system\kernel32.dll
;EXP=c:\windows\system\user32.dll
;EXP=c:\windows\system\gdi32.dll
```

Los puntos y coma al principio de algunas líneas definen comentarios que no se procesarán (al igual que el mandato REM en un fichero BAT de MS-DOS). Suprímase el punto y coma correspondiente y, si fuera necesario, modifíquese el directorio para que señale a la librería correctamente para que las cargue y queden listas para su uso. Guárdese el fichero y reinícíese el ordenador para que el cambio tenga efecto.

Se pueden añadir otras DLLs a voluntad cuyas funciones quieran emplearse como puntos de corte. Pongamos por caso que se desea depurar un programa en Visual Basic, será preciso entonces cargar el fichero msvbmXX.dll (donde XX indica la versión del lenguaje).

El fichero winice.dat puede emplearse para otros usos también (deberá utilizarse 'Symbol Loader' al no haber fichero winice.dat en Windows NT). Identifíquese la línea que comienza con INIT. Será algo parecido a:

```
INIT=„X;
```

Indica la línea de inicialización de SoftICE, esto es, una serie de mandatos que se ejecutan tras arrancar SoftICE. Gracias a ello, el usuario podrá adaptar SoftICE a sus necesidades. La siguiente línea de inicialización es utilizada por el autor:

```
lines 60;wd 10;wc 30;wl;X;
```

Los mandatos que empiezan por 'w' gobiernan el modo en que se muestran las ventanas individuales; 'lines 60' hace que el programa trabaje en ventanas de 60 líneas. El mandato 'X' resulta muy importante puesto que oculta SoftICE tras su inicialización de manera que no se muestre cada vez que se arranque. Resulta especialmente útil con Windows 9x/Me, donde SoftICE sólo se puede arrancar antes de que lo haya hecho el sistema completamente (de hecho, es SoftICE quien arranca primero para luego ejecutar Windows).

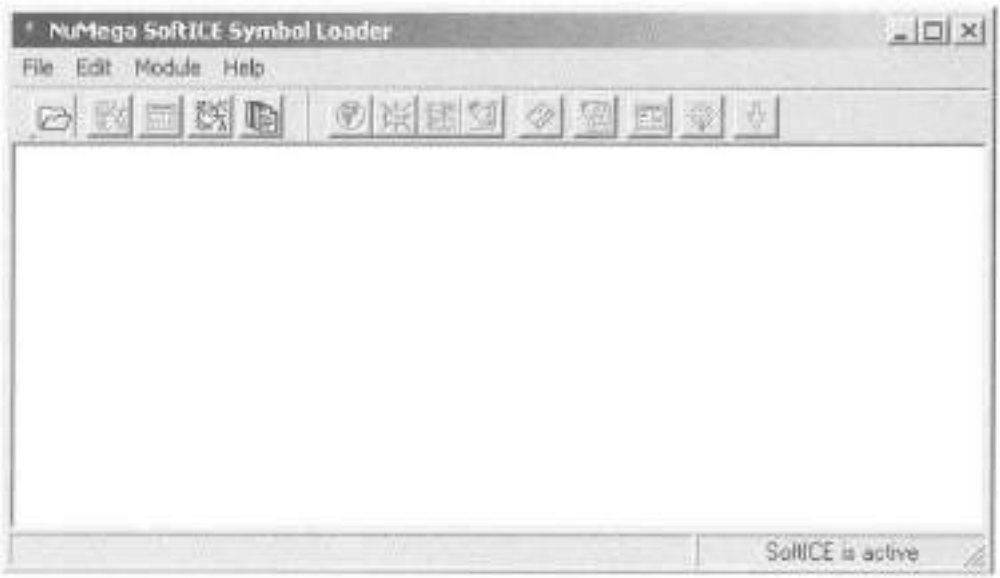

Figura 2-1. 'Symbol Loader' de SoftICE y sus propiedades

Conforme el lector se vaya haciendo familiar con SoftICE y sus mandatos, se verá capaz de configurarlo según sus necesidades.

Esto es todo lo que se necesita saber de la configuración de esta herramienta única. De producirse algún problema (por ejemplo, con los dispositivos de tarjetas gráficas), acúdase a la documentación.

Mandatos, funciones y controles básicos

WINDOWS

Lo más importante que se debe recordar es la combinación de teclas Ctrl+D para poder mostrar SoftICE (suponiendo que no se haya redefinido). Esta combinación de teclas tiene prioridad sobre las demás, no importa que ya se estén utilizando en una aplicación, juego o el mismo escritorio. Siempre conducirá a SoftICE.

En la siguiente figura se muestra una representación esquemática de SoftICE con etiquetas indicando sus ventanas básicas. Servirá de orientación al lector al utilizar el programa.

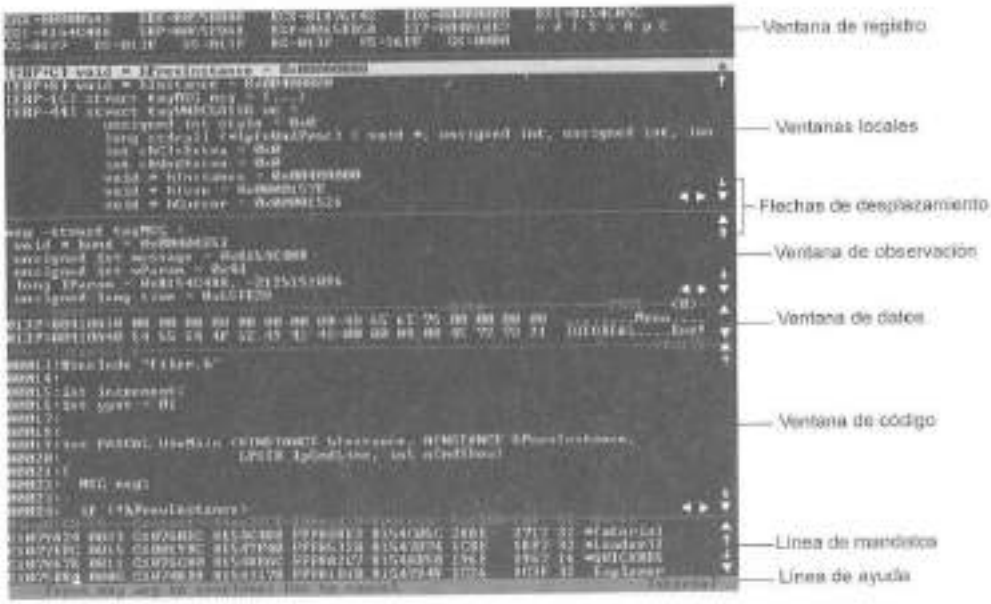

Figura 2-2. Presentación de SoftICE.

Ventana de registro ('Register window' en inglés):

La ventana de registro muestra los nombres de todos los registros y sus valores. En el margen derecho también figuran ocho letras que representan atributos (en inglés, 'flags') específicos del registro EFLAGS. Cuando el atributo queda definido se indica con el color azul. A continuación se describen los atributos:

O	D	I	S	Z	A	P	C
Overflow Flag	Direction Flag	Interrupt Flag	Siga Flag	Zero Flag (el flag más utilizado)	Auxiliary Carry Flag	Parity Flag	Carry Flag

Tanto los valores de los registros como de los atributos pueden evitarse mediante el mandato `'r valor de registro'`, o `'r fl abreviatura de atributo'`.

Ejemplo:

- `'r eax 1'` – cambia el valor del registro EAX a 1
- `'r fl z'` – cambia el estado del atributo cero (zero flag)

La ventana de registro se habilita y deshabilita con el mandato 'wr'.

Ventana de datos ('Data window', en inglés)

Bajo la ventana de registro se halla la de datos (a no ser que se activen otras funciones más avanzadas, como la ventana de locales o de observación). Mediante el mandato `'d dirección'` (se hará referencia a las direcciones de memoria en este libro por su valor hexadecimal), se podrá obtener el contenido de memoria de una dirección dada. También se podrán intercambiar las direcciones de memoria por los nombres de registro. Por ejemplo: con `'d eax'`, la ventana mostrará el contenido de memoria en la dirección correspondiente a este valor de registro. De igual manera, el contenido de la memoria se podrá editar con los mandatos: `'e ,eb ,ew ,ed ,es ,el'` y `' et'`.

El mandato `'s dirección_inicial l longitud 'cadena''` buscará cadenas en la memoria. Si la búsqueda termínase con éxito, la ventana de datos mostrará la dirección permitiendo continuar con la búsqueda mediante el mandato `'s'`.

Ejemplo:

- `'s 0 l 123456 'contraseña''` – buscará la cadena 'contraseña' en el área de memoria 0-123456. El mandato `'wd número_de_líneas'` modificará la ventana de datos así como su visualización.

Ventana de código ('Code window', en inglés)

Bajo la ventana de datos se encuentra la ventana del código que se está depurando. Con F10 y F8 se salta de una instrucción a otra; la única diferencia entre ambas radica en que con F8 SoftICE utiliza instrucciones CALL, mientras que con F10 no. Para recuperar el control desde las instrucciones CALL, o más precisamente, para detenerse tras procesar la siguiente instrucción RET, utilícese F12.

Todas estas teclas representan mandatos que el usuario puede configurar a su discreción. Si se deseara más información sobre cómo realizar esta asignación, acúdase bien al fichero winice.dat (Windows 9X/Me) o bien al 'Symbol Loader' (todas las plataformas Win32).

Ejemplo:

`'a 0040100'` mostrará el texto siguiente:

```
'XXXX:00401000 specify_the_requested_instruction'
```

Tras pulsar intro se presentará al usuario la oportunidad de editar la siguiente instrucción. Al pulsar intro de nuevo, en vez de editar una nueva instrucción, se pondrá fin a la edición. El mandato `'wc número_de_líneas'` maximizará o minimizará la ventana de datos así como su visualización.

Línea de mandatos

La última ventana es la línea de mandatos propiamente dicha, donde se ejecutan los mandatos de SoftICE. Además, incorpora el registro histórico de todos los mandatos anteriores y otra información complementaria del producto. Utilícense las siguientes teclas para desplazarse entre las diferentes ventanas (entre paréntesis el equivalente en inglés):

RePág (PageUp) – muestra la página anterior del histórico
AvPág (PageDn) – muestra la página siguiente del histórico
Flecha de mayúsculas-flecha de desplazamiento hacia arriba (Shift-arrow up) – una línea hacia arriba en el histórico
Flecha de mayúsculas-flecha de desplazamiento hacia abajo (Shift-arrow down) – una línea hacia abajo en el histórico
Alt-flecha hacia arriba (Alt-arrow up) – una línea hacia arriba en la ventana de datos
Alt-flecha hacia abajo (Alt-arrow down) – una línea hacia abajo en la ventana de datos
Alt-RePág (Alt-PageUp) – una página hacia arriba en la ventana de datos
Alt-AvPág (Alt-PageDn) – una página hacia abajo en la ventana de datos
Ctrl-RePág (Ctrl-PageUp) – una página hacia arriba en la ventana de código
Ctrl-AvPág (Ctrl-PageDn) – una página hacia abajo en la ventana de código
Ctrl-flecha hacia arriba (Ctrl-arrow up) – una línea hacia arriba en la ventana de código
Ctrl-flecha hacia abajo (Ctrl-arrow down) – una línea hacia abajo en la ventana de código

Puntos de corte

En la última parte de este capítulo se describirá lo que son los puntos de corte, su uso, las distintas formas de almacenarlos en el código de programa y de detectarlos. Ahora se indicará cómo definirlos y controlarlos.

En SoftICE, se pueden utilizar los siguientes puntos de corte:

- Punto de corte en una ejecución
- Punto de corte en el acceso a memoria
- Punto de corte en un rango de accesos a memoria
- Punto de corte en el acceso a un puerto de entrada/salida
- Punto de corte en una interrupción
- Punto de corte en un mensaje Windows

Punto de corte en una ejecución

Representa el tipo más común de punto de corte. Como su nombre indica, se practica el punto de corte en la ejecución de una instrucción dada. Si se han llegado a cargar las funciones exportadas de las DLLs correspondientes en SoftICE, también se podrán definir puntos de corte según los nombres de las funciones API. SoftICE detectará de igual modo la dirección definida por el punto de corte. Siguiendo el mismo procedimiento se podrán establecer puntos de corte comunes a las direcciones de instrucciones individuales.

Sintaxis: `'bpx dirección | nombre_de_función'`

Ejemplo:

- `'bpx MessageBoxA'` – establece el punto de corte en la función API MessageBoxA
- `'bpx 00401000'` – establece el punto de corte en la instrucción de la dirección 00401000.

Punto de corte en el acceso a memoria

Aquí el punto de corte se establece en una posición de memoria. Se asemeja mucho al anterior. Ahora bien, no sólo se puede definir en la ejecución, sino también en otras operaciones y en una cierta dirección de memoria. Los parámetros de acceso indicarán si se realiza en modo de escritura o de lectura.

Sintaxis: `'bpm dirección | nombre_de_función parámetro_de_acceso'`

A continuación se indican los parámetros de acceso:

- r – lectura
- w – escritura
- rw – lectura y escritura
- x – ejecución, el punto de corte se comportará igual que con el mandato 'bpx'

Ejemplo:

- `'bpm 00401000 r'` – define el punto de corte en modo lectura en la dirección 00401000
- `'bpm MessageBoxA x'` – define el punto de corte en la ejecución de la función API MessageBoxA

Con frecuencia los puntos de corte de este tipo se utilizan para supervisar una cierta posición de memoria y las operaciones que ahí se realizan, por ejemplo, la introducción de una contraseña, un número de serie, etc. Se suele combinar con el mandato 's' para buscar en cadenas de caracteres, lo que permitirá calcular la posición exacta del algoritmo de protección.

Punto de corte en un rango de accesos a memoria

Este tipo de punto de corte es prácticamente idéntico al anterior. La principal diferencia estriba en que los puntos de corte en un rango de accesos a memoria no sólo pueden definirse para una sola dirección, sino para un área de memoria. Estos puntos de corte no pueden emplearse en WindowsNT/2000/XP.

Sintaxis: `'bpr dirección_inicial dirección_final parámetro_de_acceso'`

Los parámetros de acceso son los mismos que los de la sección anterior:

- r – lectura
- w – escritura
- rw – lectura y escritura

Ejemplo:

- `'bpr 00401000 00402000 rw'` – define el punto de corte en modo lectura y escritura en el área de memoria comprendido entre la dirección 00401000 y 00402000.

El mandato también se puede utilizar con nombres de funciones:

- `·bpr MessageBoxA MessageBoxA+10 r'` – define el punto de corte en la lectura de los primeros diez bytes de la función API MessageBoxA.

Si bien este tipo de punto de corte se utiliza con mucha frecuencia allí donde los otros dos anteriores no son de gran provecho, parece muy difícil encontrar algún caso en el que no se pueda realizar la detección. Aunque suele resultar sencillísimo, en este libro se incluirá una introducción sobre el asunto.

Punto de corte en el acceso a un puerto de entrada/salida

Este punto de corte se realiza en un puerto concreto del ordenador.

Sintaxis: `·bpio puerto parámetro_de_acceso'`

Los parámetros de acceso podrán tener los siguientes valores:

- r – lectura
- w – escritura
- rw – lectura y escritura

Este tipo de puntos de corte resulta idóneo para, por ejemplo, anular la protección por hardware, normalmente conectada al puerto paralelo.

Ejemplo:

- `'bpio 378 r'` – define el punto de corte en lectura del puerto paralelo

Punto de corte en una interrupción

Su nombre resulta autoexplicativo.

Sintaxis: `'bpint número_de_interrupción'`

Ejemplo:

- `'bpint 3'` – define el punto de corte en el interruptor número 3.

Punto de corte en un mensaje Windows

Un punto de corte en un mensaje de Windows examina la cola de mensajes en busca del recurso definido por el manejador. Este tipo de puntos de corte se emplea frecuentemente con programas de los que no se sabe cómo generan ventanas. Para solucionarlo, ha de definirse al manejador la ventana (por ejemplo, mediante el mandato 'hwnd') y establecer puntos de corte en un mensaje concreto de Windows. Normalmente se utiliza el mensaje WM_DESTROY para suprimir la ventana de la pantalla. Al cerrar esta ventana, SoftICE mostrará un cuadro de diálogo.

Véase la parte de referencia de este libro si se desea consultar una lista de los mensajes Windows más conocidos.

Sintaxis: `'bmsg manejador mensaje_de_Windows'`

Ejemplo:

- `'bmsg 12345 WM_DESTROY'` – define el punto de corte en el recurso cuyo manejador sea 12345h y el mensaje WM_DESTROY.

GESTIÓN DE LOS PUNTOS DE CORTE

Para facilitar su uso, en SoftICE todos los puntos de corte están numerados.

Listado de los puntos de corte

Sintaxis: `'bl'`

Esta función enumera todos los puntos de corte (activos y no activos) con sus parámetros.

Ejemplo:

- `'bl'`

Mostrará algo parecido a lo siguiente:

```
00)BPX #XXXX:00012345
01)BPMB #XXXX:00006789 RW DR3
```

Supresión de los puntos de corte

Sintaxis: *'bc punto_de_corte_1, punto_de_corte_2...'*

Esta función elimina los puntos de corte señalados.

Ejemplo:

- `'bc 0'` – elimina el punto de corte número 0
- `'bc 1,4'` – elimina los puntos de corte números 1 y 4
- `'bc *'` – elimina todos los puntos de corte

Activación y desactivación de los puntos de corte

Sintaxis: *'bd punto_de_corte_1, punto_de_corte_2...'* – desactivación

'be punto_de_corte_1, punto_de_corte_2...' – activación

Ejemplo:

- `'bd 1,2,3,5'` – desactiva los puntos de corte y números 1, 2, 3 y 5
- `'be 1'` – activa el punto de corte número 1
- `'bd *'` – desactiva todos los puntos de corte
- `'be *'` – activa todos los puntos de corte.

GESTIÓN ESTRUCTURADA DE EXCEPCIONES (SEH)

Descripción y uso de la gestión estructurada de excepciones

Entre las distintas posibilidades que ofrecen en la actualidad los sistemas operativos Win32, SEH ('Structured Exception Handling', en inglés), se encuentra entre las más utilizadas, pero curiosamente, también es una de las menos documentadas. No basta con la buena documentación de mandatos como `'_try'`, `'_finally'` o `'except'`. Tan sólo constituyen servicios de librería de un lenguaje de programación. No existe la documentación debida de la SEH real incluida directamente en el sistema operativo. Escapa al conocimiento del autor la razón de la carencia de una función tan relevante, especialmente en lo que concierne a sistemas operativos tan "estables" como Windows.

El lector se preguntará qué tipo de comprobaciones y gestión de excepciones están relacionadas con el pirateo informático. Quedará sorprendido de su alto número. Resultan imprescindibles para la inmensa mayoría de los algoritmos antidepuradores de hoy en día (y también de otro tipo). Con frecuencia estos algoritmos se basan en la invocación de funciones que persistentemente intentar acceder a recursos accesibles sólo en ciertas circunstancias —normalmente cuando se detecta un problema de seguridad, por ejemplo, un depurador activo etc.— Por otro lado, cuando todo funciona normalmente, el proceso suele finalizar lanzando una excepción, que debe gestionarse. Aquí es donde SEH juega su papel. Otra propiedad de SEH es la que permite saltar de un anillo a otro (lo que se tratará posteriormente). En general, hace mucho más que comprobar excepciones. Los ejemplos resultarán más ilustrativos.

SEH en desarrollo

La estructura completa de la SEH comprende varias partes. El primer paso consiste en "instalar" nuestro propio código de comprobación y gestión de excepciones —el denominado manejador—, y guardar la dirección del anterior para recuperarlo posteriormente. Finalmente concluiremos el código cuyas excepciones vayan a comprobarse por el manejador, siguiendo una secuencia de cierre específica que recuperará el manejador SEH original. El lector podrá comprobar en el código siguiente que nuestro nuevo manejador SEH se denomina MyHandler:

```
push offset MyHandler    // se guarda la dirección de
                        //nuestro manejador definido
                       // en algún lugar del código
push dword ptr fs:[0]    // se guarda la dirección del
                        // manejador anterior
mov fs:[0],esp     // instalación del manejador nuevo
```

El código siguiente restaurará el manejador SEH original:

```
pop dword ptr fs:[0] //recupera el manejador original
add esp,4    // ajuste de pila (stack)
```

El manejador variará según las funciones y excepciones que haya de procesar y resolver.

Cada excepción que ocurra en el programa comportará una gran cantidad de información interesante y útil. Esta información se almacena en dos estructuras: CONTEXT y EXCEPTION_RECORD.

Mientras EXCEPTION_RECORD describe la excepción conforme suceda, CONTEXT incluye una lista de los datos actuales en los registros del procesador.

Será el valor devuelto por el manejador el que determine cómo procederá la ejecución del código del programa. Son dos los valores destacables:

EXCEPTION_CONTINUE_EXECUTION = 0

- La excepción quedó identificada, procesada y restaurada; el manejador ordena seguir ejecutando el programa. En caso de que no fuera restaurada, necesariamente sucederá de nuevo.

EXCEPTION_CONTINUE_SEARCH = 1

- El manejador no procesó la excepción y la entrega al manejador siguiente en la pila para su posterior procesamiento.

Los manejadores aquí utilizados no resolverán ninguna excepción complicada, sencillamente saltarán los bloques de datos con código defectuoso en la inmensa mayoría de los datos.

Las instrucciones que más recientemente se hayan procesado antes de que ocurra la excepción se almacenarán en el registro EIP, incluido, junto a otros registros, en la estructura de CONTEXT. Su tamaño aumentará según lo que la longitud exceda de lo previsto (esto es, la longitud de las instrucciones defectuosas), de manera que las instrucciones defectuosas se saltan y con ellas, la excepción. Si el manejador no resolviera la excepción de este modo y el valor obtenido fuera 0 (EXCEPTION_CONTINUE_EXECUTION), lo que indicaría que el programa continúa su ejecución, la excepción volvería a suceder.

Algoritmos comunes

ALGORITMOS BASADOS EN LA FUNCIÓN API CREATEFILEA

Seguramente, la manera más común de detectar SoftICE consista en procurar acceder a sus "drivers" (VxD en Windows 9X, SYS en WindowsNT) y librerías de todo tipo que empleen la función API CreateFileA (o su gemela para WindowsNT en Unicode CreateFileW).

Estos algoritmos se han hecho muy populares dada su increíble facilidad de uso inclusive en los lenguajes de programación de más alto nivel. Desgraciadamente, la función API CreateFileA se ha extendido tanto habida cuenta de su frecuente uso que ya no supone ningún obstáculo, ni para los principiantes, superar este tipo de detección. Razón por la que su utilidad queda prácticamente relegada si se desea disponer de un buen algoritmo de protección. Resulta sorprendente que los sistemas de protección utilizados hoy en día sigan aplicando este método histórico de detección.

Figura 2-3. Muchas protecciones evitan los depuradores

Estos métodos para detectar SoftICE se han hecho tan populares entre los crackers que la mayoría de ellos renombran todos los drivers y librerías del SoftICE que estén utilizando para no molestarse en buscar y anular estos métodos de protección (por muy fácil que resulte).

En este libro no se examinarán todos estos métodos para detectar SoftICE; por razones históricas se describirá el que probablemente sea el más famoso basado en el principio recién descrito. Conocido como MeltICE, detecta SoftICE accediendo a sus "drivers" —el "driver" VxD pertinente se denomina SICE en Windows 9x/Me y NTICE, el correspondiente "driver" SYS para la versión NT de SoftICE—. Hay otros "drivers" de SoftICE que pueden detectarse del mismo modo, por ejemplo, un "driver" con nombre SIWDEBUG u otro "driver" gráfico SIWVID.

Resulta muy sencillo:

```
/*******Windows 9x/Me*******/
HANDLE File = CreateFile("\\\\.\\SICE ",GENERIC_READ |
 GENERIC_WRITE,FILE_SHARE_READ |
 FILE_SHARE_WRITE,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL
 ,NULL);

if(File != INVALID_HANDLE_VALUE)     // ¿driver SoftICE
                                     //          localizado?
{
    CloseHandle(File);
    MessageBox("SoftICE detectado",NULL,MB_OK);
}
else
    MessageBox("SoftICE no detectado",NULL,MB_OK);

/*******Windows NT/2000/XP*******/
HANDLE File = CreateFile("\\\\.\\NTICE ",GENERIC_READ |
 GENERIC_WRITE,FILE_SHARE_READ |
 FILE_SHARE_WRITE,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,
 NULL);
if(File != INVALID_HANDLE_VALUE)     // ¿driver SoftICE
                                     //          localizado?
{
    CloseHandle(File);
    MessageBox("SoftICE detectado",NULL,MB_OK);
}
else
    MessageBox("SoftICE no detectado",NULL,MB_OK);
```

LA INTERFAZ BOUNDSCHECKER Y EL USO DE INIT3

El siguiente es uno de los procedimientos habituales de detectar SoftICE. Emplea su interfaz, que permite control parcial, incluso desde otros programas. La interfaz más conocida de este tipo se denomina BCHK (BoundsChecker); otra también bien conocida es la llamada FGJM. Como en el caso anterior, se persigue acceder a la interfaz de SoftICE. Si se negara el acceso, SoftICE no estaría activo. De lo contrario, SoftICE sí estaría activo (o bien se habría producido algún error inesperado —pudiéndose comprobar entonces el valor de retorno para asegurarse—), y así quedarían accesibles con otros fines las funciones propias de BoundsChecker. A la interfaz BoundsChecker la invoca la interrupción INIT3, el número de función solicitada por el registro EAX y otros valores "mágicos" de los registros pertinentes. Por esta razón, la instrucción INT3 no resulta procesada por el sistema sino por SoftICE.

SoftICE sólo procesará la invocación a INT3 si está activo. En caso contrario, el sistema procesará INT3, con lo que se generará una excepción STATUS_BREAKPOINT que deberá ser reparada por nuestro servicio SEH. Éste es el modo de saber si SoftICE está o no activo. Si a INT3 no le preceden valores apropiados en los registros, la invocación siempre será procesada por el sistema (a no ser que el mandato i3here de SoftICE esté activo —algo que se estudiará más adelante—). Esta propiedad de SEH constituye también un método para generar excepciones, aunque, como se indicará más adelante en la sección sobre puntos de corte, esta llamada a la interrupción resulta muy fácil de detectar. Razón por la que ha de considerarse simplemente como un ejemplo que otra excepción sustituya (y hay muchas donde elegir).

SEH también puede controlar la comprobación de excepciones. Puede aplicarse la popular función API `SetUnhandledExceptionFilter` a expensas de degradar considerablemente la seguridad de los mecanismos de protección. Ya se ha comentado el uso innecesario de las funciones API en los mecanismos de protección y se volverá a insistir en el asunto en este capítulo también, en la sección dedicada a puntos de corte.

Para comprobar si SoftICE está activo, guardaremos el valor mágico BCHK = BoundsChecker en el registro EBP, y el valor 4 el registro EAX. El código resultante será parecido al siguiente:

```
__asm
{
    push offset MyHandler              // guardando dirección
                                          de nuestro
                                          manejador
    push dword ptr fs:[0]              // guardando dirección
                                          del manejador
                                          anterior
    mov fs:[0],esp                     // instalación
    push ebp                           // valor EBP guardado
    mov ebp,4243484Bh                  // EBP ={ASCII}BCHK =
                                          BoundsChecker
    mov eax,4                          // EAX contiene la
                                          función
                                          solicitada
    INT 3                              // Llamada a
                                          BoundsChecker - si
                                          SoftICE no está
                                          activo, el sistema
                                          procesa la llamada,
                                          con lo que
                                          MyHandler procesará
```

```
                                    la excepción
                                    resultante
    nop                             // NOP por
                                    compatibilidad con
                                    sistemas Win32 -
                                    cuando INT3 causa la
                                    excepción, EIP
                                    (NT/2000/XP) señala
                                    a INT 3, pero
                                    EIP(9x/Me), a  NOP
    pop ebp                         // recuperación de EBP
    pop dword ptr fs:[0]            // recuperación del
                                    manejador anterior
    add esp,4                       // ajuste de pila
    cmp eax,4                       // si SoftICE procesa
                                    INT3, EAX cambiará
                                    de valor
    jz No_SoftICE                   // en caso contrario
                                    el sistema procesará
                                    INT3 o habrá
                                    sucedido una
                                    excepción inesperada
    jmp SoftICE

MyHandler:
                                    // para facilitar el
                                    acceso se leen las
                                    estructuras CONTEXT
                                    y EXCEPTION_RECORD
                                    en e.g. ECX y EDX
    mov edx,[esp+0Ch]               // CONTEXT
    mov ecx,[esp+4]                 // EXCEPTION_RECORD
    mov eax,80000003h               // EAX = valor de la
                                    excepción provocada
                                    por la instrucción
                                    INT3-
                                    STATUS_BREAKPOINT
                                    (80000003h)
    mov ecx,[ecx]                   // ECX = número de
                                    excepciones
                                    generadas
    sub eax,ecx                     // comparación de los
                                    valores de EAX y ECX
    je Ok                           // seguir si los
                                    valores de las
                                    excepciones son
                                    idénticos
```

```
    sub eax,eax                          // otra excepción, que
                                            no se procesará por
                                            manejador
    inc eax                            // ExceptionContinueSearch
    jmp End

Ok:
    add dword ptr [edx+0B8h],1           // el valor EIP se
                                            amplía 1 byte,i.e.,
                                            la longitud de la
                                            instrucción INT3
                                            (CCh)
                                         // así se saltará la
                                            instrucción y se
                                            evitará la excepción
    sub eax,eax                        // ExceptionContinueExecution

End:
    ret
}
No_SoftICE:
    MessageBox("SoftICE no detectado",NULL,MB_OK);
    return;

SoftICE:
    MessageBox("SoftICE detectado",NULL,MB_OK);
```

Si se optara por una aplicación más fácil haciendo uso de la función API SetUnhandledExceptionFilter, el código podría ser el siguiente:

```
long __stdcall MyHandler(_EXCEPTION_POINTERS
                                        *ExceptionInfo)
{
    if (ExceptionInfo->ExceptionRecord->ExceptionCode ==
    0x80000003)
    {
        ExceptionInfo->ContextRecord->Eip +=1;
                                                  // excepción
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    return EXCEPTION_CONTINUE_SEARCH;
}

void ...::SomeFunction ()
```

```
{
    LPTOP_LEVEL_EXCEPTION_FILTER Previous =
    SetUnhandledExceptionFilter(MyHandler);
    __asm
    {
        push ebp                        // se guarda el valor
                                           EBP
        mov ebp,4243484Bh               // EBP =(ASCII)BCHK =
                                           BoundsChecker
        mov eax,4                       // EAX contiene la
                                           función solicitada
        INT 3                           // llamada a
                                           BoundsChecker -
                                           MyHandler procesará
                                           una excepción si
                                           SoftICE no estuviera
                                           activo
        nop
        pop ebp                         // se recupera EBP
        cmp eax,4
        jz No_SoftICE
    }
    MessageBox("SoftICE detectado",NULL,MB_OK);
    SetUnhandledExceptionFilter(Previous);
    return;

No_SoftICE:
    MessageBox("SoftICE no detectado",NULL,MB_OK);
    SetUnhandledExceptionFilter(Previous);
}
```

Acúdase a la sección de referencia del libro si se desearan conocer otras funciones de la interfaz BoundsChecker y su uso. Si aún resultase insuficiente, en el manual denominado "Ralf Brown's Interrupt List" incluido en el CD adjunto, se podrá encontrar una lista de todas las interrupciones para todo tipo de programas comunes o insospechados.

La manera de detectar el depurador TRW resulta prácticamente idéntica. La única diferencia radica en que no es necesario guardar ningún valor mágico en los registros antes de la invocación a INT3. Si TRW no estuviera activo, se generaría una excepción.

EMPLEO DE INT1

Si se sustituye INT1 por INT3, no se precisa guardar ningún otro valor en los registros, al igual que con la detección de TRW. Desgraciadamente, este método sólo funciona con los sistemas operativos NT, 2000 y XP, lo que a su vez exige detectar el sistema operativo:

```
mov eax,cs
test al,100b
jne Win9x                          // salir si el sistema
                                      operativo fuera
                                      Windows 9x/Me

o:
push fs:[30h]
pop eax
test eax,eax
js Win9x                           // salir si el sistema
                                      operativo fuera
                                      Windows 9x/Me

o:
mov ax,ds
test al,4
jne Win9x                          // salir si el sistema
                                      operativo fuera
                                      Windows 9x/Me
```

Cuando se utiliza INT1, el manejador debe ser un poco más sofisticado, puesto que esta interrupción provoca una excepción incluso aunque SoftICE se encuentre activo. Si así fuese, INT1 causará la excepción `STATUS_SINGLE_STEP = 80000004h`; y en caso contrario, la excepción `STATUS_ACCESS_VIOLATION = C0000005h`. Ha de ponerse también cuidado con el valor del registro EIP. Cuando SoftICE esté activo, EIP señalará la instrucción siguiente a INT1, con lo que no se tendrán que ajustar nada manualmente. Si, por el contrario, SoftICE no estuviera activo, EIP señalará la instrucción que provocó la excepción —esto es, INT1—. Ésta es la causa de que el registro EIP deba ampliarse en 2 bytes (la longitud de la instrucción INT1 - CD01h). El código resultante se asemejará al siguiente:

```
__asm
{
    mov ax,ds
    test al,4                       // identificación del
                                       sistema operativo
    jne Win9x                       // salir si el sistema
                                       operativo fuera
                                       Windows 9x/Me

    xor eax,eax                     // EAX = 0
    push offset MyHandler           // guardando la
                                       dirección de nuestro
                                       manejador
    push dword ptr fs:[eax]         // guardando la
                                       dirección del
                                       manejador anterior
mov fs:[eax],esp                    // instalación
    INT 1
    pop dword ptr fs:[0]            // recuperación del
                                       manejador anterior
    add esp,4                       // ajuste de pila
    cmp eax,4                       // SoftICE detectado?
    jz No_SoftICE
    jmp SoftICE
MyHandler:
    mov edx,[esp+0Ch]               // CONTEXT
    mov ecx,[esp+4]                 // EXCEPTION_RECORD
    mov eax,80000004h               // EAX = valor de la
                                       excepción
                                       STATUS_SINGLE_STEP
                                       causado por INT1
                                       cuando SoftICE está
                                       activo
    mov ecx,[ecx]                   // ECX = valor de la
                                       excepción recién
                                       generada
    sub eax,ecx                     // comparación de los
                                       valores de EAX y ECX
    je Ok                           // si los valores de
                                       las excepciones son
                                       idénticos, SoftICE
                                       está activo
    sub ecx,0C0000005h              // si SoftICE no está
                                       activo, INT 1
                                       provocada la
                                       excepción
```

```
                                        STATUS_ACCESS_VIOL
                                        ATION (C0000005h)
    je No_NTIce                      // ¿ es la excepción
                                    STATUS_ACCESS_VIOLATION?
    sub eax,eax                        // se génerо un error
                                       y no será procesado
                                       por este manejador
    inc eax                         // ExceptionContinueSearch
    jmp End

No_NTIce:
    add dword ptr [edx+0B8h],2         // se restaura la
                                       excepción saltando
                                       la instrucción
                                       incorrecta
    mov [edx+0B0h],4                   // 4 guardado en EAX -
                                       SoftICE no detectado

Ok:
    sub eax,eax                     // ExceptionContinueExecution
End:
    ret
}
Win9x:
    MessageBox("Este método sólo funciona con Windows
     NT/2000/XP",NULL,MB_OK);
    return;

No_SoftICE:
    MessageBox("SoftICE no detectado",NULL,MB_OK);
    return;

SoftICE:
    MessageBox("SoftICE detectado",NULL,MB_OK);
```

Figura 2-4. Armadillo resulta muy sobrio cuando detecta un depurador

EMPLEO DE INT 68H

Por el contrario, la detección de un depurador empleando la interrupción INT 68h, para lo que no se requiere SEH, funciona sólo con Windows 9x/Me. El sistema siempre procesa INT 68h y no suceden excepciones.

```
__asm
{
    mov ax,ds
    test al,4                          // identificación del
                                          sistema operativo
    je Not_Win9x                       // fin si el sistema
                                          operativo fuera
                                          Windows 9x/Me
    mov ah,43h                         // AH = número de
                                          servicio
    INT 68h
    cmp ax,0F386h                      // si el valor
                                          devuelto fuera otro
                                          número "mágico"
                                          F386h, SoftICE
                                          estaría activo
    jz SoftICE
}
    MessageBox("SoftICE no detectado",NULL,MB_OK);
    return;

SoftICE:
    MessageBox("SoftICE detectado",NULL,MB_OK);
    return;

Not_Win9x:
    MessageBox("Este método sólo funciona con Windows
     9x/Me",NULL,MB_OK);
```

BÚSQUEDA EN EL REGISTRO DE WINDOWS

Un método muy simple y eficaz para detectar la instalación de SoftICE en un ordenador dado consiste en buscar por el registro de Windows. Se incluye aquí con ánimo de ser exhaustivos puesto que este método, al igual que sucedía con el basado en el uso de `CreateFileA`, no supone ningún obstáculo para un cracker experimentado. La mayoría de ellos han renombrado o modificado de alguna manera todas las definiciones hechas en el registro por SoftICE para impedir el funcionamiento de este método de detección. Por otro lado, este método sólo permitirá detectar la instalación de SoftICE en un ordenador

concreto, y no si está activo. Por eso debiera considerarse el resultado de esta comprobación como una señal de advertencia, y no la clave de la protección. Resulta lícito solicitar al usuario que inactive el depurador mientras se utilice un programa, pero no que lo desinstale. El mismo problema afecta a los mismos métodos de detección con otros programas (desensambladores, editores hexadecimales, etc.) razón por la que no se ha incluido en este libro.

```
HKEY Key;
BYTE VerDataBuffer[200],InstDataBuffer[200];
DWORD VerSize = 5,InstSize = 128;

if (RegOpenKeyEx(HKEY_LOCAL_MACHINE,
 "Software\\NuMega\\SoftICE\\",NULL,KEY_READ,&Key) ==
 ERROR_SUCCESS) // ¿se detectó
                          // la clave añadida por SoftICE?
{
    RegQueryValueEx(Key,"Current
     Version",NULL,NULL,VerDataBuffer,&VerSize);
                                          // SoftICE version
    RegQueryValueEx(Key,"InstallDir",NULL,NULL,
     InstDataBuffer,&InstSize);      // directorio de
                                    // instalación de SoftICE
    MessageBox((const char*)InstDataBuffer,(const
     char*)VerDataBuffer,MB_OK);
}
else
    MessageBox("SoftICE no detectado",NULL,MB_OK);
```

Figura 2-5. SoftICE detectado

BÚSQUEDA EN AUTOEXEC.BAT

En los sistemas operativos Windows 9x/ME, el fichero autoexec.bat arranca SoftICE. Lo que facilita enormemente su detección mediante la búsqueda en dicho fichero de la referencia a winice.exe. En este caso se ha empleado MFC (Microsoft Foundation Class) para trabajar con cadenas de caracteres:

```
CFile File;
char SIString[] = "winice.exe";
CString String;
BOOL SoftICE = FALSE;

if (File.Open("c:\\autoexec.bat",CFile::modeRead) &&
 File.GetLength() != 0)
{
    BYTE *pMem = new BYTE [File.GetLength()+1];
                                        // asignación de memoria
    File.Read(pMem,File.GetLength()+1);     // carga del
                                        // fichero en memoria

    String = pMem;
    String.MakeLower();                 // conversión de todas
                                           las letras a
                                           minúscula

for (int i = 0;i < String.GetLength()-
 strlen(SIString);i++)
{                                       // búsqueda gradual de
                                           la secuencia por
                                           todo el fichero
    if (String.Mid(i,strlen(SIString)) == SIString)
                                        // cadera detectada
    {                                   // ¿„winice.exe "?
        SoftICE = TRUE;
        break;
    }
}
    delete []pMem;
}
if (SoftICE)
    MessageBox("SoftICE detectado",NULL,MB_OK);
else
    MessageBox("SoftICE no detectado",NULL,MB_OK);
```

Con mucha frecuencia se encuentran algoritmos que hacen uso de los vectores de interrupción. Generalmente se considera que estos algoritmos son los mejores puesto que no utilizan ninguna función API y, por tanto, no resultan fácilmente detectables ni identificables. Sin embargo, la realidad es bien distinta. No constituye ningún problema identificar cualquier intento de utilizar IDT (véase 'Privilegios de anillo' o el capítulo 4 - FrogsICE) por parte de un programa, encontrar su protección y anularla.

PUNTOS DE CORTE

La definición de puntos de corte constituye una parte crucial de cada depurador. No es más que un juego de mandatos con sus parámetros que ejecuta el depurador para interrumpir la ejecución de un programa y permitir su depuración según las circunstancias.

En la inmensa mayoría de los casos, los puntos de corte se definen en llamadas a funciones API por estar bien documentadas y, sobre todo, por el hecho de que se utilizan con mucha frecuencia por programadores y crackers. Las funciones API, por tanto, son el mejor punto de entrada a los programas.

Ya se ha señalado en el capítulo 1 (donde se puede encontrar una lista exhaustiva de las funciones API mejor conocidas) el uso de las funciones API en los algoritmos de protección. Después de haber presentado en las anteriores secciones los distintos tipos de puntos de corte que se pueden definir con SoftICE, apenas resulta necesario señalar lo fácil, en mayor o menor medida, que resulta para un cracker identificar dónde se ubica el algoritmo de protección en un programa. Éste será el sitio idóneo para que los puntos de corte apunten a funciones del propio programa y vayan penetrando hasta el corazón de la protección y del programa.

Por lo tanto, téngase en mente que cualquier uso de una función API en un mecanismo de protección reduce su seguridad global. Lo primero que un cracker intentará emplear al buscar la ubicación del algoritmo de protección serán las funciones API, cuya invocación resulta fácilmente identificable utilizando puntos de corte en un depurador.

El lector podrá apreciar que con los depuradores, los puntos de corte constituyen el peligro número uno; por eso se debe prevenir su uso.

Ya se han examinado los distintos tipos de puntos de corte en la sección sobre el uso del SoftICE. Se puede dividir según se dirijan al software o al hardware. Los primeros se "guardan" directamente en el software, mientras que los de hardware se guardan directamente en la CPU, en los denominados registros de depuración. Los puntos de corte para software se definen mediante mandatos `bpx`, `bpr` y `bpint`, mientras que los de hardware se definen con los mandatos `bpm` y `bpio`.

Puntos de corte para software

PUNTOS DE CORTE EN UNA INTERRUPCIÓN (BPINT)

Conforme se viene indicando, muchos algoritmos de protección se centran principalmente en evitar el uso de SoftICE invocando diversos tipos de interrupciones. Lo que pudiera llevar a pensar que los puntos de corte sobre interrupciones suponen una

solución ideal. No ocurre así realmente. No debe olvidarse que un punto de corte se define sobre una interrupción del sistema, no sobre una interrupción de SoftICE. El punto de corte sólo se podrá utilizar cuando sea el sistema el que realice la interrupción independientemente de que SoftICE esté activo. (¿Cómo se podría establecer el punto de corte si no estuviese activo?). Así sucede con INT 68h; en caso contrario, los puntos de corte de esta índole no resultan útiles. Por eso no deben utilizarse interrupciones para, por ejemplo, generar excepciones con SEH si se desea mantener la seguridad.

Resulta pertinente mencionar a este respecto dos mandatos de SoftICE muy parecidos al uso de puntos de corte en interrupción, concretamente `i1here ON/OFF` e `i3here ON/OFF`. Los resultados obtenidos al activar estos mandatos son casi idénticos a los puntos de corte bpint sobre las interrupciones INT1 e INT3. La única diferencia estriba en que SoftICE procesa las llamadas a la interrupción, con lo que no sucede ninguna excepción. Resultan utilísimos cuando se desea depurar cierto lugar de un programa sin necesidad de buscarlo durante mucho tiempo. Basta con realizar la invocación en una de las interrupciones (normalmente a INT3) en el lugar indicado y definir el mandato oportuno.

PUNTOS DE CORTE EN UNA EJECUCIÓN (BPX)

SoftICE (y otros depuradores) sobrescriben las secciones del programa donde se definen puntos de corte en ejecución con el valor INT3 (una interrupción bastante obvia), esto es, CCh. Así, obtiene la información de que se va a interrumpir la ejecución del programa. Cuando se ejecute el código del programa, todas las secciones que se hayan modificado de esta manera se volverán a escribir con el valor original y así evitaremos un error en el programa. De este modo, sólo será preciso comprobar los bytes del código elegido o del área de memoria, etc., sobrescritos en memoria a CCh, esto es, INT3, para encontrar un punto de corte en una ejecución. Se puede alegar, con razón, que el valor CCh del código no sólo resulta válido para INT3, sino que puede formar parte de cualquier otra instrucción. Por eso, este método debiera utilizarse preferentemente para comprobar código del que se sabe con certeza que no contiene el valor CCh en las áreas de memoria que se van a examinar. Este método se torna ineficaz para comprobar grandes áreas de código desconocido —resulta preferible utilizar comprobaciones de integridad de los datos ('checksums', en inglés)— (véase el capítulo 6).

Este método resulta idóneo cuando se desea realizar una comprobación rápida y sencilla de la presencia de puntos de corte bpx en funciones comunes o importantes en el código. Es exactamente todo lo que se precisa para comprobar las funciones API.

En el siguiente ejemplo se comprueban los cinco primeros bytes de la función API `DialogBoxParamA` de un punto de corte bpx:

```
HMODULE Handle = GetModuleHandle("user32");  // imageBase
                                             // user32

FARPROC Address =
 GetProcAddress(Handle,"DialogBoxParamA");
                                        // dirección de la
                                           función
                                           DialogBoxParamA

byte Function[5];
memcpy(&Function,Address,sizeof Function);   // función =
                                           los 5 primeros bytes
                                           de la función

for (int i = 0;i < 5;i++)
{
    if (Function [i] == 0xCC)              // ¿ detectado punto
                                           de corte bpx?
    {
        MessageBox("Punto de corte BPX
         detectado",NULL,MB_OK);
        return;
    }
}
MessageBox("Punto de corte BPX no detectado",NULL,MB_OK);
```

No sólo se puede obtener la dirección de la función mediante la función API `GetProcAddress`. Se puede obtener de igual manera empleando el formato PE (y las funciones importadas). En la sección dedicada al formato PE del capítulo 7 se abundará más en esta cuestión.

PUNTOS DE CORTE EN UN ÁREA DE MEMORIA (BPR)

Desde que los puntos de corte bpx y bpm dejaron de ser una novedad hace tiempo, los crackers se han dado prisa en aprender otras alternativas: los puntos de corte bpr. Resulta bastante sorprendente que aún no los haya detectado ningún sistema de protección a pesar de que su realización sea mucho más simple que con los puntos de corte bpm.

Para poder detectar los puntos de corte bpr, es necesario saber primero, cómo se definen con SoftICE.

Al no definirse sobre una única dirección los puntos de corte de este tipo, sino sobre un área de memoria dada, el procedimiento difiere del descrito anteriormente. El método utilizado por SoftICE es mucho más sofisticado. Se basa en la modificación de los derechos de acceso a aquellas páginas de memoria que incluyen áreas en las que se han definido puntos de corte. Los privilegios de acceso se modifican a PAGE_NOACCESS, con lo que cualquier acceso a las páginas señaladas de memoria (en lectura, escritura,

ejecución) causará una excepción. SoftICE la detectará y la corregirá. A continuación calculará qué tipo de acceso se realizó y lo comparará con el que se indicó en el punto de corte para el área de memoria definida. Si todas estas comprobaciones resultan idénticas, el programa se detendrá, el punto de corte habrá cumplido su cometido.

No resulta nada complicado definir los puntos de corte de este algoritmo de detección. Se basa sencillamente en la comprobación de los derechos de acceso de un área de memoria específica. Si el derecho de acceso se define como PAGE_NOACCESS, el punto de corte bpr quedará establecido en el área de memoria indicada.

A continuación se muestra un ejemplo de protección mediante puntos de corte bpr:

```
MEMORY_BASIC_INFORMATION Mbi;
DWORD Address;

__asm
{
BprTest:
    mov Address,offset BprTest        // dirección de
                                         comienzo de la
                                         comprobación

}
VirtualQuery ((void*)Address,&Mbi,sizeof
 MEMORY_BASIC_INFORMATION);

if (Mbi.Protect == PAGE_NOACCESS)     // ¿derechos de acceso
                                         = PAGE_NOACCESS?
    MessageBox("Punto de corte BPX detectado", NULL,MB_OK);
else
    MessageBox("Punto de corte BPX no detectado",
    NULL,MB_OK);
```

La función VirtualQuery identifica los parámetros de la página de memoria donde resida la dirección de comienzo de la comprobación, además aporta información de todas las páginas adyacentes en memoria con los mismos parámetros (algo complejo hecho fácil). Si se definiese un punto de corte bpr en este área de memoria, el algoritmo lo detectaría con seguridad.

Puntos de corte hardware

Como ya se comentó anteriormente, los registros de depuración de la CPU se encargan de guardar directamente los puntos de corte hardware. Se denominan "hardware"

por la ubicación donde se guardan (directamente en la CPU, frente a los puntos de corte "software", que se guardan en el software), y también por ser principalmente el procesador, y no sólo la herramienta de depuración (como sucede con los puntos de corte software) el encargado de su procesamiento. Al resultar idéntico el almacenamiento de todos los tipos de puntos de corte hardware disponibles en SoftICE, también (ambos) se podrán detectar del mismo modo.

Empléese el código siguiente si se desea depurar registros definidos en la estructura CONTEXT como dobles palabras Dr0-Dr3, Dr6 y Dr7:

```
typedef struct _CONTEXT {
...
DWORD Dr0;
DWORD Dr1;
DWORD Dr2;
DWORD Dr3;
DWORD Dr6;
DWORD Dr7;
...
}CONTEXT;
```

Los puntos de corte como tales pueden guardarse en los registros Dr0 a Dr3. Dr7 es un registro de mandato del sistema de depuración de la CPU y Dr6 es un registro de estado.

Si el programa estuviera en ensamblador, pruébese el siguiente acceso para depurar los registros:

```
mov eax,Dr1
```

Sucederá una excepción de forma natural puesto que estas instrucciones sólo se pueden ejecutar en ciertos anillos privilegiados del procesador (que se describirán posteriormente). Si bien es cierto que se puede cambiar de uno a otro para poder ejecutar el ejemplo anterior, constituye un procedimiento innecesariamente largo que sólo funciona con Windows 9x/Me y que apenas confiere más robustez al programa (por no mencionar los puntos débiles que acarrea esta técnica de depuración mediante la detección del contenido de los registros).

Resulta preferible sacar partido al SEH clásico. Ya se mencionó en el capítulo correspondiente, que las excepciones quedan asociadas a dos estructuras, EXCEPTION_RECORD y CONTEXT, y que la última contiene una lista de los registros de la CPU, y no sólo los comunes, como EAX, EBX, etc., también los de depuración (como se acaba de describir).

Desgraciadamente, nada resulta tan fácil como parece: no basta con provocar una excepción y comprobar el contenido del registro de depuración en la estructura CONTEXT del manejador —esto es, comprobar que es igual a cero (sin puntos de corte definidos)—. Al menos no si se va a utilizar esta técnica con todas las plataformas Win32. Parece que si este método se aplica a Windows NT/2000/XP —a diferencia de Windows 9x/Me— el contenido de los registros de depuración obtenidos en el manejador SEH quedan modificados con valores incorrectos.

Razón que obliga a elegir un método ligeramente diferente. La aplicación primero rellena los registros de depuración con sus propios datos (direcciones) con los que se establecerán los puntos de corte hardware. Lo que resulta equivalente de hecho a definir los puntos de corte en la propia aplicación (quedará parcialmente contemplado en el programa). A continuación comprobará que se hayan procesado todos los puntos de corte definidos (esto es, que hayan causado la excepción STATUS_BREAKPOINT) y en caso contrario, que se hayan sobrescrito por los puntos de corte hardware definidos por el depurador.

Para provocar la excepción que necesita el manejador SEH, se suele emplear la típica instrucción INT3. Como ya se indicó anteriormente, una excepción provocada de esta manera sólo tiene valor pedagógico. Resulta muy sencillo detectarla, debe sustituirse el procedimiento de protección por algún otro método para generar excepciones difícil de detectar. Por tanto, no debe hacerse referencia a la función API RaiseException.

Resulta mucho mejor experimentar insistentemente con los ejemplos de puntos de corte hardware: los que saltan de anillo y otros semejantes. Además, no sólo se pueden emplear para detectar puntos de corte. Se pueden borrar o sobrescribirlos con ánimo de detectar el depurador o para muchas otras operaciones. Los puntos de corte hardware son interesantes y algunas veces, incluso enigmáticos en ciertos aspectos que merece la pena estudiar.

DESCRIPCIÓN DE UN PROGRAMA DE EJEMPLO EMPLEADO PARA DETECTAR PUNTOS DE CORTE HARDWARE

El corazón de todo el programa radica en un manejador denominado BPMHandler. Primero repara la excepción generada intencionadamente (por la instrucción INT3). La estructura rellenada con datos tras la generación de la excepción se emplea para definir los puntos de corte para la aplicación. Puesto que los puntos de corte causan excepciones en otros programas, el manejador se encargará también de capturarlas. Se utilizará un solo registro de los puntos de corte tratados para guardar la información en la variable Done. Son cuatro los puntos de corte definidos (registros Dr0, 1, 2 y 3). Si el valor de la variable Done fuera también 4, los puntos de corte habrán finalizado con éxito. En caso contrario, el depurador estaría activo en memoria y se habrían definido algunos puntos de corte hardware. Por último, el manejador restaura la última excepción del programa causada por la segunda instrucción INT3, cuya tarea consiste en eliminar el

registro de mandato Dr7. Al definir los puntos de corte en la aplicación, el registro de mandato Dr7 debe completarse con los valores necesarios. Si se deseara saber por qué el registro recoge el valor 155h, acúdase a la descripción de los registros Dr6 y 7 recogida en el CD adjunto. Aunque ya supondría adentrarse en un área más avanzada que no resulta imprescindible en este momento.

Igualmente debe ponerse cuidado en el orden en el que se definen los puntos de corte. Puesto que el procesamiento de los registros de depuración se inicia en Dr3 hasta llegar a Dr0, el primer punto de corte emparejado con la primera sentencia NOP, debe guardarse en el registro Dr3 y el último en el registro Dr0.

```
DWORD OurAddress;
BYTE Done = 0,DR7Break = 0;

__asm
{
    push offset BPMHandler          // guardando la
                                       dirección de nuestro
                                       manejador
    push dword ptr fs:[0]           // guardando la
                                       dirección del
                                       manejador anterior
    mov fs:[0],esp                  // instalación

    mov OurAddress,offset Address;  // OurAddress -
                                       dirección de
                                       arranque para los
                                       puntos de corte
    INT 3                           // primera excepción
                                       intencionada creada
                                       por INT3
                                    // instrucción elegida
                                       para simplificar la
                                       construcción del
                                       manejador (como el
                                       lector habrá
                                       observado), no se
                                       emplee en la
                                       protección bajo
                                       ningún concepto
    nop
    nop                             //---I primer punto de
                                       corte
    nop                             //   I
```

```
                                    //   I
    nop                             //---I segundo punto
                                       de corte
    nop                             //   I
                                    //   I- ubicaciones
                                       para puntos de corte
    nop                             //---I tercer punto de
                                       corte
    nop                             //   I
                                    //   I
Address:                            //   I
    nop                             //---I cuarto punto de
                                       corte
    nop                             //

    inc DR7Break                    // ampliando el valor
                                       de la variable
                                       DR7Break..., vacía
                                       el registro Dr7 en
                                       el manejador

    INT 3                           // segunda excepción
                                       intencionada para
                                       limpiar el registro
                                       Dr7
    nop

    pop dword ptr fs:[0]            // recuperación del
                                       manejador anterior
    add esp,4                       // ajuste de pila
    jmp Jump

BPMHandler:
    mov edx,[esp+0Ch]               // CONTEXT
    mov ecx,[esp+4]                 // EXCEPTION_RECORD
    push ebp                        // guardando el valor
                                       EBP
    mov ebp,[edx+0B4h]              // la lectura del
                                       valor previo EBP
                                       antes de introducir
                                       el manejador
                                       facilita el uso de
                                       nuestras variables
    mov ebx,OurAddress
    mov [edx+4],ebx                 // Dr0
    sub ebx,2
    mov [edx+8],ebx                 // Dr1
```

```
    sub ebx,2
    mov [edx+0Ch],ebx                  // Dr2
    sub ebx,2
    mov [edx+10h],ebx                  // Dr3 - contiene el
                                          punto de corte del
                                          primer NOP
    and [edx+14h],0                    // limpiando el
                                          registro Dr6
    mov [edx+18h],155h                 // definiendo el valor
                                          del registro de
                                          mandato Dr7
    cmp DR7Break,0                     // ¿fue causa de la
                                          segunda excepción de
                                          la instrucción INT
                                          3?
    je MYjump
    and [edx+18h],0                    // en caso afirmativo
                                          se limpia el
                                          registro de mandato
                                          Dr7

MYjump:
    mov eax,80000004h                  // EAX = valor de la
                                          excepción
                                          STATUS_SINGLE_STEP,
                                          causada por el punto
                                          de corte definido
    mov ecx,[ecx]                      // ECX = valor de la
                                          excepción recién
                                          producida
    sub eax,ecx                        // comparación de los
                                          valores EAX y ECX
    je BreakPoint                      // si los valores de
                                          las excepciones
                                          resultan idénticos,
                                          aumentamos el número
                                          de puntos de corte
                                          para tratarlos
                                          satisfactoriamente
    dec eax                            // ¿fue causa de la
                                          excepción
                                          STATUS_BREAKPOINT
                                          (80000003h) la
                                          instrucción INT 3?
    je Ok                              // en caso negativo,
                                          sucedió otra
                                          excepción de
```

```
                                        programa, que no
                                        tratará este
                                        manejador
    sub eax,eax
    inc eax                  // ExceptionContinueSearch
    jmp End

BreakPoint:
    inc Done                        // aumentando el
                                       número de puntos de
                                       corte tratados
                                       correctamente

Ok:
    add dword ptr [edx+0B8h],1      // reparando la
                                       excepción saltando
                                       la instrucción
                                       incorrecta
    sub eax,eax              // ExceptionContinueExecution

End:
    pop ebp                         // se recupera EBP
    ret

Jump:
}

if (Done == 4)                      // ¿establecidos todos
                                       los puntos de corte
                                       satisfactoriamente?
    MessageBox("Puntos de corte hardware no detectados",
     NULL,MB_OK);
else
    MessageBox("Puntos de corte hardware detectados",
     NULL,MB_OK);
```

MÉTODOS AVANZADOS

Privilegios de los anillos

Un procesador puede funcionar en cuatro niveles diferentes según el número de instrucciones privilegiadas que ejecute. Se denominan: anillo 0 (mayores privilegios), anillo 1, anillo 2 y anillo 3 (con menores privilegios sucesivamente). La mayoría de los programas se ejecutan en el anillo 3 y, por lo tanto, cuentan con más limitaciones. Como ya se habrá dado cuenta el lector por los ejemplos empleados con los puntos de corte hardware, resulta imposible escribir en los registros de depuración. Existen programas, sin embargo, que sí pueden ejecutarse directamente en el anillo 0. Entre ellos los drivers lógicos VxD y los drivers de dispositivo SYS en Windows 9x/Me y sólo los de tipo SYS en Windows NT/2000/XP (los de tipo VxD no pueden emplearse con Windows NT/2000/XP). Hay tres formas, que se sepa, de saltar entre el anillo 3 y el anillo 0 directamente desde el programa: mediante LDT, IDT y SEH. Y que están prohibidos en Windows NT/2000/XP por razones de seguridad, sólo se podrán utilizar con Windows 9x/Me. De hecho, hay un método más de invocar el código del anillo 0 desde una aplicación normal: utilizando la función ordinal de la librería kernel32.dll, `Kernel32!ORD_0001`. (En el capítulo 7 se describirá en qué consiste la función ordinal —formato PE—. Por ahora basta con saber que es una función sin nombre.) Como de costumbre, la función sólo está disponible en Windows 9x/Me, su invocación exige un largo procedimiento, mucho más complicado que el procedimiento habitual para saltar al anillo 0, y su capacidad, más reducida, no resulta muy útil. Bien es cierto que detectar este método resulta algo más difícil, pero lo es mucho más el basado en SEH. Por eso no se describirá en este libro, bastará con constatar su existencia indicando que se aplica principalmente en VxDCalls, que se describirá a continuación. En cierto sentido es triste que una técnica tan eficaz, como la de saltar al anillo 0, sólo pueda emplearse en la práctica con Windows 9x/Me (esto es, no resulta viable programar, por ejemplo, un driver SYS propio), reduciendo así la eficacia del número de mecanismos disponibles en los sistemas NT.

MANERAS DE SALTAR ENTRE EL ANILLO 3 Y EL ANILLO 0

Lista de descriptores locales (LDT)

Probablemente constituya el método más antiguo de saltar del anillo 3 al anillo 0. No merece la pena detenerse en comentar el código, innecesariamente complicado, anticuado y en desuso. Ha quedado sustituido por los restantes dos métodos, mucho mejores. Se mencionarán aquí sólo con ánimo de ser exhaustivos.

```
BYTE Gdt[6];
Gdt[0] = 0;
```

```
Gdt[1] = 0;
Gdt[2] = 0;
Gdt[3] = 0;
Gdt[4] = 0;
Gdt[5] = 0;

BYTE Call_Ring0[6];
Call_Ring0[0] = 0;
Call_Ring0[1] = 0;
Call_Ring0[2] = 0;
Call_Ring0[3] = 0;
Call_Ring0[4] = 0xF;
Call_Ring0[5] = 0;

BYTE Callgate[8];
Callgate[0] = 0;
Callgate[1] = 0;
Callgate[2] = 0x28;
Callgate[3] = 0;
Callgate[4] = 0;
Callgate[5] = 0xEC;
Callgate[6] = 0;
Callgate[7] = 0;

DWORD Gate = (DWORD)Callgate;

_asm
{
    mov ax,ds
    test al,4                          // ¿sistema operativo?
    je Not_Win9x                       // bifurcación si el
                                          sistema operativo no
                                          es Windows 9x/Me

Win9x:
    mov eax,offset Ring0               // EAX = dirección de
                                          la parte del
                                          programa que se
                                          ejecutará en el
                                          anillo 0
    mov word ptr [Callgate],ax         // dirección del
                                          código que se
                                          ejecutará en el
                                          anillo 0 en nuestro
                                          callgate

    shr eax,16
    mov word ptr [Callgate+6],ax
```

```
    xor eax,eax
    sgdt fword ptr Gdt                  // se guarda GDT
    mov ebx,dword ptr [Gdt+2]           // EBX = dirección de
                                           GDT

    sldt ax
    add ebx,eax                         // obtención de la
                                           dirección del
                                           descriptor

    mov al,[ebx+4]
    mov ah,[ebx+7]
    shl eax,16                          // EAX = dirección de
                                           LDT
    mov ax,[ebx+2]                      // obtención de la
                                           dirección del
                                           descriptor callgate

    add eax,8
    mov edi,eax                         // EDI = dirección en
                                           callgate para
                                           aplicar cambios
    mov esi,Gate                        // ESI = dirección de
                                           nuestra callgate
    movsd                               // se sobrescribe el
                                           anterior callgate

    movsd

    call fword ptr [Call_Ring0]         //--> anillo 0

    xor eax,eax
    sub edi,8
    stosd                               // recuperación de la
                                           anterior callgate

    stosd
    jmp Ok

    /*******Anillo 0*******/
Ring0:
    mov eax,Dr7                         // esta interrupción
                                           sólo funcionará en
                                           el anillo 0
    retf                                //-->Anillo 3
}
Not_Win9x:
    MessageBox("Éste método sólo funciona con Windows
     9x/Me",NULL,MB_OK);
    return;
```

```
Ok:
    MessageBox("Cambio de anillo satisfactorio",
    NULL,MB_OK);
```

Lista de descriptores de interrupciones (IDT)

Representa otro método bien conocido para que el programa salte del anillo 3 al anillo 0. Esta técnica se basa en la *sustitución de vector*, esto es, la sustitución de una rutina con ciertos interruptores. La interrupción se ejecutará en el anillo 0 y si dirigiéramos una rutina de interruptores a nuestro propio código, también se ejecutará en el anillo cero.

La sustitución de vector se utiliza abundantemente en algoritmos antidepuradores de todo tipo. Se redirigen las rutinas de interrupción que precisa el depurador (INT1 e INT3), lo que complica muchísimo e incluso evita su uso. Resulta, sin embargo, necesario considerar que, puesto que el sistema y algunas otras aplicaciones también utilizan algunas interrupciones con frecuencia, no se pueden sobrescribir sin más. Siempre habrá de restaurar su estado original, sin olvidarse tampoco de la multitarea, el sistema está constantemente alternando entre los procesos en ejecución (aunque el usuario los perciba como simultáneos): si se desea preservar la integridad, deberá restaurarse la rutina de interrupción original antes de que el sistema salte a otro proceso. Resulta decisivo recuperar la interrupción pertinente.

Aunque este método no sea malo, no llega a alcanzar al siguiente basado en SEH. A causa, en primer lugar, de los problemas con la redefinición de interrupciones recién mencionados, y en segundo, por su fácil detección (véase el capítulo 4, FrogsICE).

El siguiente programa salta del anillo 3 al anillo 0 utilizando IDT y la interrupción INT3:

```
_asm
{
    mov ax,ds
    test al,4                          // ¿sistema operativo?
    je Not_Win9x                       // bifurcación = el
                                          sistema operativo no
                                          es Windows 9x/Me

Win9x:
    push edx
    sidt [esp-2]                       // carga de IDT en la
                                          pila
    pop edx
```

```
    add edx,(3/*=INT 3*/*8)+4          // EDX - vector de la
                                          interpretación
                                          seleccionada, i.e.
                                          INT 3
    mov ebx,[edx]
    mov bx,word ptr [edx-4]            // guardando la
                                          dirección de la
                                          rutina de
                                          interruptores para
                                          restaurarla
                                          posteriormente
    lea edi,Ring0
    mov [edx-4],di
    ror edi,16                         // redirección de la
                                          rutina de
                                          interruptores al
                                          código propio que
                                          vaya a ejecutarse en
                                          el anillo 0
    mov [edx+2],di
    push ds                            // guardando valores
                                          importantes de los
                                          registros
    push es
    INT 3                              //-->Anillo 0
    pop es                             // recuperación de los
                                          registros
    pop ds
    mov [edx-4],bx                     // recuperación de la
                                          anterior rutina de
                                          interruptores
    ror ebx,16
    mov [edx+2],bx
    jmp Ok

    /*******Ring0*******/
Ring0:
    mov eax,Dr7                        // esta instrucción
                                          sólo funcionará en
                                          el anillo 0
    iretd                              //-->Anillo 3
}
Not_Win9x:
    MessageBox("Este método sólo funciona con Windows
     9x/Me",NULL,MB_OK);
    return;
```

```
Ok:
    MessageBox("Cambio de anillo satisfactorio",
    NULL,MB_OK);
```

Gestión estructurada de excepciones (SEH)

Éste es el tercer y último método para procurar que un programa salte del anillo 3 al anillo 0. Es el más difícil de detectar. El programa primero causa una excepción, que dispara al manejador, lo que brinda la posibilidad de ajustar todos los registros y valores elegantemente, de manera que al devolver el control el manejador al programa, éste se ejecuta en el anillo cero.

Merece la pena destacar, que si bien el método resultaba muy difícil de detectar, no debe emplearse la instrucción INT3 para crear la excepción. Algo, que a estas alturas, el lector debe ya conocer.

```
_asm
{
    mov ax,ds
    test al,4                       // ¿sistema operativo?
    je Not_Win9x                    // bifurcación = el
                                       sistema operativo no
                                       es Windows 9x/Me

Win9x:
    push offset MyHandler           // guardando la
                                       dirección de nuestro
                                       manejador

    push dword ptr fs:[0]           // guardando la
                                       dirección del
                                       manejador anterior
    mov fs:[0],esp                  // instalación
    pushfd                          // guardando EFLAGS
    mov eax,esp                     // guardando el valor
                                       ESP, esto es, la
                                       dirección de la pila

    INT 3                           //-->Anillo 0
                                    // innecesario aquí
                                       NOP - el código se
                                       ejecuta sólo en
                                       Windows 9x//Me
```

```
                                    // el manejador no
                                       necesita restaurar
                                       la excepción (
                                       saltándola modifica
                                       el valor EIP)
                                       provocada, ya que
                                       EIP, en Windows
                                       9x/Me señala a la
                                       instrucción
                                       siguiente a INT 3

    /*******Ring0*******/
    mov ebx,Dr7                     // esta instrucción
                                       sólo funcionará en
                                       el anillo 0

                                    // recuperación de los
                                       valores anteriores
                                       de los registros
                                       antes de saltar al
                                       anillo 3
    push edx                        // GS
    push edx                        // FS
    push edx                        // ES
    push edx                        // DS
    push edx                        // SS
    push eax                        // ESP
    push dword ptr [eax]            // EFLAGS
    push ecx                        // CS
    push offset Ring3               // EIP = dirección del
                                       código donde el
                                       programa se
                                       ejecutará en el
                                       anillo 3 de nuevo
    iretd                           //-->Anillo 3

Ring3:
    popfd                           // recuperación de
                                       EFLAGS
    pop dword ptr fs:[0]            // recuperación del
                                       manejador anterior
    add esp,4                       // ajuste de pila
    jmp Ok
```

```
    /*******MyHandler*******/
                                        // manejador SEH para
                                           saltar al anillo 0
MyHandler:
    mov edx,[esp+0Ch]                   // CONTEXT
    mov ecx,[esp+4]                     // EXCEPTION_RECORD
    mov ecx,[ecx]                       // ECX = valor de la
                                           excepción recién
                                           ocurrida
    cmp ecx,80000003h                   // ¿causó INT3 la
                                           excepción
                                           STATUS_BREAKPOINT
                                           (80000003h)?
    jne Error                           // bifurcación = se
                                           produjo otra
                                           excepción que no
                                           procesará este
                                           manejador
    movzx ecx,word ptr [edx+0BCh]       // ECX = CS
    mov [edx+0ACh],ecx                  // ECX(tras la
                                           bifurcación desde el
                                           manejador, vuelta al
                                           código del programa)
                                           = CS
    mov dword ptr [edx+0BCh],28h        // CS(tras la
                                           bifurcación ...) =
                                           28h, para saltar al
                                           anillo 0
    movzx ecx,word ptr [edx+0C8h]       // ECX = SS
    mov [edx+0A8h],ecx                  // EDX(tras la
                                           bifurcación ...) =
                                           ECX = SS
    mov dword ptr [edx+0C8h],30h        // SS(tras la
                                           bifurcación...) =
                                           30h, para saltar al
                                           anillo 0
    or dword ptr [edx+0C0h],200h        // EFLAGS(tras la
                                           bifurcación...) =
                                           200h
    sub eax,eax                         //
                                      ExceptionContinueExecution
    ret                                 //--> anillo 0

Error:
    sub eax,eax
    inc eax                             //
                                     ExceptionContinueSearch
```

```
    ret
}
Not_Win9x:
    MessageBox("Este método sólo funciona con Windows
     9x/Me",NULL,MB_OK);
    return;

Ok:
    MessageBox("Cambio de anillo satisfactorio",
     NULL,MB_OK);
```

Detección de SoftICE mediante VxDCall

El próximo método de detección de un SoftICE activo resulta muy semejante al que emplea la función API `CreateFileA`. De nuevo se intentará acceder a los drivers de SoftICE, en este caso mediante VxDCall. Estas llamadas las realizan los drivers lógicos (librerías VxD) únicamente desde el anillo 0. Por esta razón, el método sólo se puede utilizar con Windows 9x/Me. Se invocará la función `VMMCall Get_DDB` para saber si un driver específico se ha guardado o no en memoria.

A un driver se le identifica o por su nombre o por su identificador. Aquí se eligirá el identificador: 202h, para el driver denominado SICE. También se pueden utilizar otros drivers para la detección, por ejemplo, el driver gráfico denominado SIWVID, cuyo identificador es 7A5Fh. Cuando la detección sea difícil, se aplicará el método SEH para saltar al anillo 0.

```
DWORD ID = 0x202;                    // SICE
// DWORD ID = 0x7A5F;                // SIWVID
DWORD Name = 0;                      // innecesario definir
                                        nombre

_asm
{
    mov ax,ds
    test al,4                        // ¿sistema operativo?
    je Not_Win9x                     // bifurcación = el
                                        sistema operativo no
                                        es Windows 9x/Me

Win9x:
    push offset MyHandler            // guardando la
                                        dirección de nuestro
                                        manejador
```

```
    push dword ptr fs:[0]           // guardando la
                                       dirección del
                                       manejador anterior
    mov fs:[0],esp                  // instalación

    pushfd
    mov eax,esp
    INT 3                           //--> anillo 0
                                    // NOP innecesario
                                       aquí ...

    /*******Anillo 0*******/
    push edx
    push edx
    push edx
    push edx
    push edx
    push eax
    push dword ptr [eax]
    push ecx
    push offset Ring3

    mov eax,ID                      // EAX = identificador
                                       del driver
    mov edi,Name                    // EDI = nombre del
                                       driver

    /*******VxDCall*******/
    __emit 0xCD                     // VMMCall Get_DDB
    __emit 0x20                     //
    __emit 0x46                     //
    __emit 1                        //
    __emit 1                        //
    __emit 0                        //

    test ecx,ecx                    // ¿detectado driver?
    jz Not_Detected
    xor eax,eax                     // en caso afirmativo,
                                       EAX = 0

Not_Detected:
    iretd                           //--> Anillo 3

Ring3:
    popfd
    pop dword ptr fs:[0]            // recuperación del
                                       manejador anterior
```

```
    add esp,4                          // ajuste de pila

    test eax,eax                       // ¿driver detectado?
    jnz No_SoftICE
    jmp SoftICE

/*******MyHandler*******/
                                       // manejador SEH para
                                          saltar al anillo 0
MyHandler:
    mov edx,[esp+0Ch]                  // CONTEXT
    mov ecx,[esp+4]                    // EXCEPTION_RECORD
    mov ecx,[ecx]                      // ECX = valor de la
                                          excepción recién
                                          ocurrida
    cmp ecx,80000003h                  // ¿causó INT3 la
                                          excepción
                                          STATUS_BREAKPOINT
                                          (80000003h)?
    jne Error                          // bifurcación = se
                                          produjo otra
                                          excepción que no
                                          procesará este
                                          manejador
    movzx ecx,word ptr [edx+0B8h]
    mov [edx+0ACh],ecx
    mov dword ptr [edx+0BCh],28h
    movzx ecx,word ptr [edx+0C8h]
    mov [edx+0A8h],ecx
    mov dword ptr [edx+0C8h],30h
    or dword ptr [edx+0C0h],200h
    sub eax,eax            // ExceptionContinueExecution
    ret                                //--> anillo 0

Error:
    sub eax,eax
    inc eax                // ExceptionContinueSearch
    ret
}
Not_Win9x:
    MessageBox("Este método sólo funciona con Windows 9x/Me
    ",NULL,MB_OK);
    return;

No_SoftICE:
    MessageBox("SoftICE no detectado",NULL,MB_OK);
    return;
```

```
SoftICE:
    MessageBox("SoftICE detectado",NULL,MB_OK);
```

Desactivación de la tecla de atención de SoftICE

En último lugar se describe este método para detectar un SoftICE activo, un verdadero placer para un autor cuyo apodo es "trapflag". Este método consiste en intentar localizar los drivers de SoftICE que controlan su tecla de atención para mostrar el programa (Ctrl+D por omisión). Si los drivers se localizan, la instrucción correspondiente se sobrescribirá y la tecla de atención quedaría desactivada (además, se informa mediante un mensaje que SoftICE se ha desactivado, se aconseja suprimirlo y no informar al usuario de nada). Con este propósito se utiliza una librería VxD (aquí no resulta muy sensato utilizar VxDCall, tampoco resulta práctico emplear un driver SYS con Windows NT/2000/XP), que restringe el uso del método a Windows 9x/Me. Otro obstáculo radica en la presencia de la propia librería VxD. Una librería de este tipo dentro del programa junto a la desactivación de la tecla de atención de SoftICE, constituye una pista para que el cracker sepa lo que ha sucedido. Es más, puesto que todo el control de la librería se ciñe a su llamada a la función API `CreateFileA`, se podría suprimir con sencillez. Resulta necesario enmascarar la librería y, si fuera posible, su contenido. Resulta de gran ayuda el que la aplicación utilice librerías VxD para otros cometidos. (Elíjase un nombre apropiado para la librería, y no algo como ¡AntiSoftICE.VxD!) La mejor manera consiste en integrar la función dentro de otra librería VxD del programa, así el cracker que suprima la invocación a la librería mediante la función API `CreateFileA` provocará una excepción distinta en el programa.

A continuación se muestra el código de la librería VxD (original de 'trapflag' con comentarios traducidos):

```
; detector y supresor de SOFTICE by trapflag
; esta vxd detectará softice en memoria y deshabilitará
; su  manejador de la tecla de atención - actualmente
; con dos NOPs, así sice será aún accesible ( mediante
; puntos de corte y  etc.) también se puede parchear
; con EBFE..
; si vas a utilizar mi código en tus programas,
; indícalo y envíame una copia de tus programas.
; saludos a the_owl por su ayuda.
; trapflag@backtrace.de

.486p
include vmm.inc
include vwin32.inc
include shell.inc
```

```
DECLARE_VIRTUAL_DEVICE
DBG,1,0,DBG_Control,UNDEFINED_DEVICE_ID,UNDEFINED_INIT_O
RDER
; mensajes que se deben tratar
Begin_control_dispatch DBG
     Control_Dispatch Sys_Dynamic_Device_Init,
     OnDeviceInit Control_Dispatch
     Sys_Dynamic_Device_Exit,
     OnDeviceDestroy Control_Dispatch
     w32_DeviceIoControl,OnDeviceIoControl
End_control_dispatch DBG

; ------------------------------------------------
; Segmento de datos bloqueado
; ------------------------------------------------

VxD_LOCKED_DATA_SEG

; algunas variables

sequence db 0e4h,060h    ; código (opcode) 60 del
                         ; "manejador de la tecla de
                         ; atención" de SoftICE

Message db '¡SoftIce detectado y suprimido!',0
Caption db 0
pszName db 80 dup(0)
scasb_exception Exception_Handler_Struc <>

VxD_LOCKED_DATA_ENDS

; ------------------------------------------------
; Segmento de código bloqueado
; ------------------------------------------------

VxD_PAGEABLE_CODE_SEG

BeginProc OnDeviceInit          ; si el usuario carga el
                                ; dispositivo int 3

    mov scasb_exception.EH_Reserved,0
    mov scasb_exception.EH_Start_EIP,offset32
    scasb_protect
    mov scasb_exception.EH_End_EIP,offset32
    scasb_endprotect
    mov scasb_exception.EH_Handler,offset32 EHandler
```

```
    mov     esi, offset32 scasb_exception
    VMMCall Install_Exception_Handler     ; manejador de
                                ; excepciones para tratar
                                ; los errores de página
    cld
    mov edi,0C0000000h          ; dirección base
                                ; para la región
                                ; compartida del sistema
                    ; aquí se suspenden las vxds ...
    mov ecx,0FFFFFFh        ; exploración de los
                               ;primeros 000FFFFFFh bytes
seek:
mov     al,01dh                 ; búsqueda de 1dh
mov     esi,offset32 sequence  ;esi= apunta a opcodes del
                    ; manejador de la tecla de atención
scasb_protect:
        repne scasb             ; búsqueda de [edi]
scasb_endprotect:
        test  ecx,ecx           ;counter=0 ?
        jz    exit              ;salir
        push  edi               ;guardar edi
        push  ecx               ;e incluso ecx
        add   edi,4             ;edi=edi+4
        mov   ecx,2             ;compara 2 bytes de [esi]
                               ; a [edi]
        repe  cmpsb             ; "
        pop   ecx               ;recupera el mayor mr ecx
        pop   edi               ;¿su secuencia relativa no
                               ;fue el manejador de la
        jnz   seek              ; tecla de  atención de
                               ;sice?
                               ;-> buscar
        add   edi,4             ; secuencia de byte/
                               ;manejador detectado
                               ;int 3
       mov   word ptr [edi],09090H  ; dando un toque
      ; maligno al manejador de la tecla de atención
       xor ebx, ebx             ; manejo máquina virtual
       mov eax, MB_SYSTEMMODAL      ; flags de mensajes
       mov ecx, OFFSET32 Message    ; dirección del
                                   ; mensaje
       mov edi, OFFSET32 Caption    ; dirección de
                                   ;caption
       VxDcall SHELL_SYSMODAL_Message ; mostrar mensaje
                                     ; -> sice detectado
```

```
exit:
;int 3
        clc
        ret                          ; volver al cargador

EHandler:
;int 3
        test ecx,ecx                 ;¿ fin de la búsqueda?
        jz exit                      ;sí, salir
        dec ecx                      ;no, entonces contador dec
        inc edi                      ;inc puntero
        jmp scasb_protect ; seguir buscando

EndProc OnDeviceInit

BeginProc OnDeviceDestroy      ; si el usuario descarga
                               ; el dispositivo


        clc
        ret
EndProc OnDeviceDestroy

BeginProc OnDeviceIoControl          ;DEBE manejarse
        assume esi:ptr DIOCParams
        .IF [esi].dwIoControlCode == DIOC_Open
             xor   eax,eax
        .ENDIF
        ret
EndProc OnDeviceIoControl


VxD_PAGEABLE_CODE_ENDS


End
```

OTROS USOS SENCILLOS DE SEH

En el ejemplo anterior se describía cómo detectar y desactivar la tecla de atención de SoftICE. Existen otras alternativas de enfrentarse al problema. Dicha combinación de teclas también se puede emplear para otros objetivos. La tecla de atención que invoca al programa SoftICE sustituye a todas las otras teclas de atención. Si en una aplicación se asignase una función importante a esta tecla de atención que además no pudiera arrancarse de otro modo (por ejemplo, mostrar la ventana de registro de un programa), se ejecutaría SoftICE en vez de la función de dicha aplicación y, por tanto, no se podría depurar nada.

No representa ningún obstáculo modificar la teclas de atención (tampoco en el caso de Windows NT/2000/XP, puesto que SoftICE se puede arrancar como un servicio y no antes del propio Windows como sucede con Windows 9x/Me), si bien resulta bastante incómodo. A continuación se muestra el código:

```
if((GetAsyncKeyState(VK_CONTROL) &
 0x8000)&&(GetAsyncKeyState('D') &
 0x8000)&&(GetActiveWindow() == this))
{
                                        // función importante
                                           - por ejemplo,
                                           arranque de la
                                           ventana de registro
                                           de la aplicación
}
```

Definase la función para que procese el mensaje `WM_TIMER`. Sólo restará definir el contador mediante la función `SetTimer` (por ejemplo, en la función `OnInitDialog()`):

```
SetTimer(1234,100,NULL);                // definición de
                                           contador
```

No hay que creerse que métodos primitivos como éste detendrán a un cracker, pero al menos le contrariarán un poco. Aún no ha visto este autor llevar a la práctica este método de protección. No obstante, existe un programa en Internet cuya tecla de atención "secreta" se revela tras pagar por registrarse. No es una mala idea.

El lector habrá podido apreciar en los modelos anteriores algunas propiedades muy interesantes de SEH.

Por ejemplo, la modificación del registro EIP (en el manejador), que sirve para sortear la instrucción incorrecta en nuestro ejemplo, puede utilizarse para bifurcaciones mucho más largas en vez de tratar las excepciones: bifurcaciones "imperceptibles" en el código del programa. Únicamente ha de enmascararse la operación dentro del código y así poder saltar de una función a otra sin que nadie lo aprecie a primera vista (naturalmente no en un código de varias líneas).

Si se tratase de depurar el ejemplo dado para detectar el punto de corte hardware, se observará algo realmente interesante. En el momento en el que el depurador llega a una de las instrucciones NOP donde se define el punto de corte, el depurador se detiene —queda "bloqueado" de algún modo—. Razón por la que se ha de considerar la definición de

puntos de corte en instrucciones NOP no críticas repartidas por el código con ánimo de dificultar más la depuración. Aún queda algún otro consejo que aportar antes de proceder con el siguiente capítulo.

No debe nunca olvidarse que aunque se utilice SEH en beneficio propio, también se puede emplear en nuestra contra. Afirmación que queda bien ilustrada en el caso de las protecciones comerciales Armadillo y VBox. En ambos casos combinan la invocación a INT3 con la función API `SetUnhandledExceptionFilter` para detectar el depurador activo. Sus autores, obviamente, no invirtieron mucho tiempo ni esfuerzo en el diseño de la protección, puesto que el manejador no distingue ninguna excepción en particular. Se asume que la causa de todas las excepciones sea la instrucción INT3, esto es, depurador inactivo. Pero, ¿qué sucedería si alguien causara una excepción adrede? Los autores del sistema de protección no consideraron esta posibilidad, así que sólo resta causar una excepción en el programa para, paradójicamente, sortear su mecanismo de protección. En conclusión: siempre habrán de extremarse las precauciones con el manejador y nunca olvidar verificar la idoneidad (¡verdadera ironía!) de las posibles excepciones y comprobar los valores obtenidos, por ejemplo, al invocar INT3 en combinación con la interfaz BoundsChecker.

CAPÍTULO 3

PROTECCIÓN CONTRA LOS DESENSAMBLADORES

Ya se ha definido anteriormente el concepto de desensamblaje; se trata de analizar un programa hasta revertirlo a su código ensamblador original. Resulta especialmente útil al tratar de entender la lógica de ciertos algoritmos. También sirve para orientarse dentro del programa y, sobre todo, para buscar cadenas de caracteres, funciones importadas, posiciones en un programa e identificar dónde se invocan y cómo se emplean.

La causa principal de la gran difusión de los desensambladores radica en su capacidad para buscar dentro de un programa cadenas de caracteres y funciones importadas. Gracias a estas propiedades, un cracker puede aproximarse casi inmediatamente al algoritmo de protección y estudiarlo sin dificultar y, por consiguiente, anularlo. Todo ello convierte a los desensambladores en armas tremendamente potentes, incluso en las manos de un principiante.

DESENSAMBLADORES HABITUALES

Desde el advenimiento del cracking, existen dos programas prácticamente considerados estándares en el campo del desensamblaje: *W32Dasm* e *IDA*. Por supuesto, hay muchos otros, pero *W32Dasm* e *IDA* son sencillamente los

desensambladores más habituales hoy en día. A pesar de que W32Dasm se ha quedado anticuado, de que se abandonó su desarrollo y de que puede parecer inmanejable en la práctica, los crackers aún lo utilizan con mucha frecuencia. Lo aceptaron de tal modo que aunque en realidad se haya paralizado su desarrollo, ellos continúan desarrollando mejoras, actualizaciones y parches (incluidos en el CD adjunto). Realmente constituye un ejemplo perfecto de cómo aplicar la ingeniería inversa para mejorar un programa.

W32Dasm resulta bastante accesible para todo tipo de usuarios. Aunque no se pueda personalizar mucho el proceso de desensamblaje y configuración, para compensar, el programa resulta muy sencillo e intuitivo, con una buena presentación del código ensamblador resultante. El programa incluye un cuadro de referencias de cadenas y funciones importadas, una utilidad para buscar texto, y muchas otras propiedades que facilitan enormemente la búsqueda y el análisis de un programa. Una vez aplicado el mantenimiento, el programa incorporará otras características, como un editor hexadecimal. En la siguiente sección se describe el uso de esta herramienta.

IDA ('Interactive DisAssembler', en inglés) constituye, como su nombre indica, una herramienta completa de desensamblaje profesional totalmente configurable. Rey indiscutible de los desensambladores, con sus opciones y características, no tiene rival en su campo. La increíble flexibilidad de este producto (con la posibilidad de programar en un lenguaje semejante a C) permite desensamblar correctamente programas inasequibles para W32Dasm. El proceso de desensamblaje también se puede adaptar y controlar manualmente de muchos modos. Su flexibilidad se demuestra, sin embargo, en las modificaciones que el usuario puede realizar durante el proceso, lo que a primera vista podría parecer algo complejo y enrevesado. El programa está destinado para usuarios más avanzados y no para principiantes.

USO ELEMENTAL DE W32DASM

Su entorno de trabajo parece muy organizado y bien dispuesto. El primer paso consiste en elegir el fichero que se desea desensamblar: selecciónese la opción "Open File to Disassemble" del menú "Disassembler". Tras desensamblar el fichero elegido (su duración dependerá del tamaño del fichero), quedarán activadas otras opciones del programa.

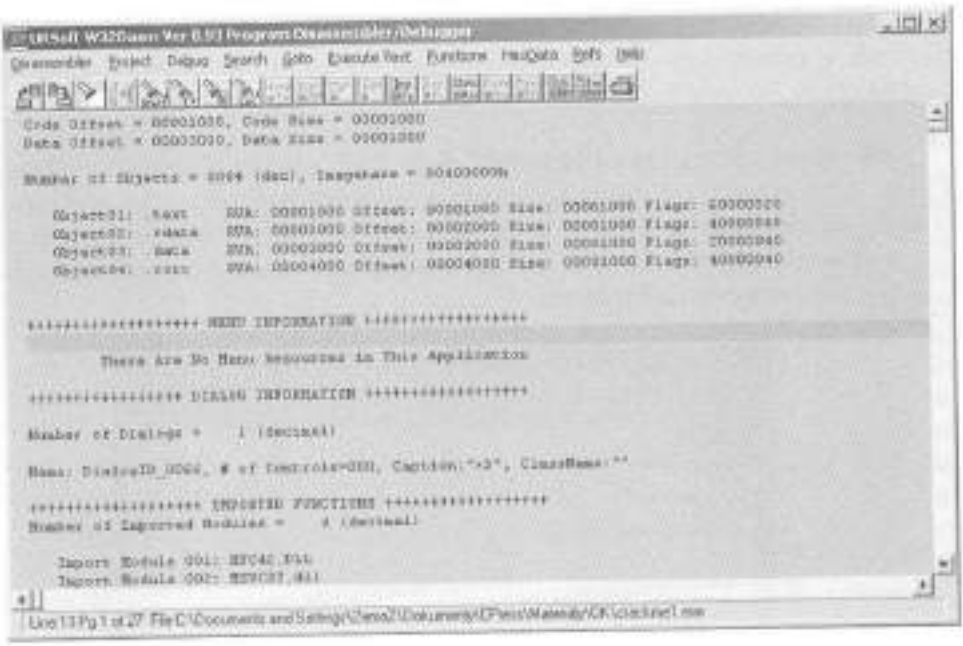

Figura 3-1. Entorno de trabajo del desensamblador W32Dasm

Pulsando la tecla F10, o el botón "Goto Program Entry Point", se accede al punto de entrada del código desensamblado del programa (véase el capítulo 7 sobre el formato PE si se desea más detalles). Igualmente se puede acceder a cualquier otra dirección pulsando la tecla de mayúsculas y F12 o bien el botón "Goto Code Location".

Se puede acceder a la lista de funciones importadas pulsando el botón "Imports", y a la lista de funciones exportadas pulsando el botón "Exports".

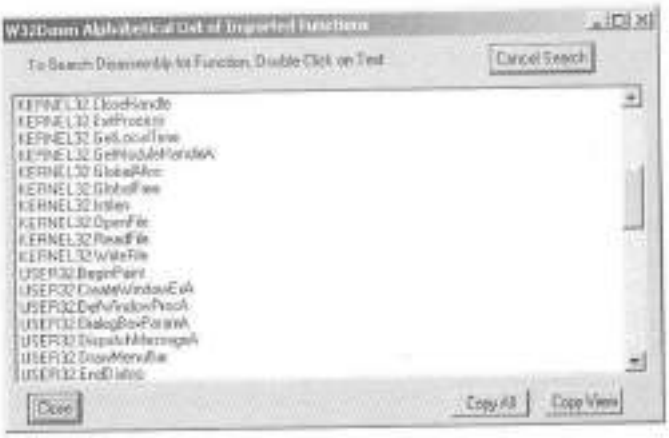

Figura 3-2. Lista de funciones importadas

Cuando se pulsa en una función API, se accede a la ubicación dentro del programa desde donde se realiza su invocación. Si esta función se invocase varias veces, al pulsar en ella seguidamente, se recorrerían todas las direcciones de acceso a la función.

Selecciónese "String Data References" si se desea obtener la lista de cadenas de caracteres. Aquí se pueden encontrar cadenas muy interesantes que conducen directamente al corazón del algoritmo de protección. Si se tratase de desensamblar un programa de código compartido, probablemente se hallaría una indicación con datos sobre el registro incorrecta o correctamente realizado, etc.

Figura 3-3. Lista de cadena de caracteres muy sugerente

Las cadenas de caracteres se manejan de forma muy similar a como se hace con las funciones importadas: basta con pulsar en cualquier cadena para saltar al código desensamblado que la emplee de algún modo.

Resulta muy sencillo analizar y trabajar con el propio código del programa desensamblado. Las referencias a instrucciones tipo "jump" y "CALL" quedan destacadas en el código, de modo que resulta muy fácil orientarse incluso con los algoritmos más complicados. Una franja azul destaca la instrucción presente.

Pulsando la flecha hacia la derecha se pueden rastrear los saltos condicionados y no condicionados, así como las instrucciones "CALL" (o bien pulsando el botón "Jump to", o en el caso de las instrucciones CALL, el botón "Call"). Pulsando la fecha hacia la izquierda (o la tecla intro) se vuelve a la instrucción CALL. Para los saltos utilícese Control+ flecha hacia la izquierda (o el botón "Ret JMP").

En caso de trabajar frecuentemente con ficheros grandes cuyo desensamblaje consume mucho tiempo, resulta más práctico guardar los ficheros en código de ensamblado. Para ello, selecciónese la opción "Save Disassembly text File and Create

Project File" del menú "Disassembler". Siempre se podrá cargar un fichero desde el menú "Project" con la opción "Open Project File".

Con esto termina la introducción a las características básicas de W32Dasm. Por su sencillez y comodidad, el lector podrá descubrir por sí mismo otras características del programa. No se describirá aquí su uso como depurador, puesto que ni se acerca a la calidad de SoftICE.

ALGORITMOS COMUNES

Al igual que sucedía con los mecanismos antidepuradores, los programas han de protegerse contra algunas de las características de los desensambladores o contra el desensamblaje en su totalidad. Si con el primero el programa se defendía detectando depuradores activos, puntos de corte, etc. (en este capítulo se introducirán algunos métodos antidepuradores que dificultarán aún más la tarea al depurador), con los mecanismos antidesensamblaje no se podrá proceder de igual manera. La razón de ello estriba en que, al igual que sucede cuando se edita el código de un programa con un editor hexadecimal, el programa no se ejecuta y, por tanto, sólo puede protegerse pasivamente. Aunque sea posible la detección activa (por ejemplo, buscando los nombres de los procesos, ventanas, etc. en ejecución), resulta prácticamente estéril (eso sin considerar el hecho de que, a diferencia de SoftICE, existen muchísimos desensambladores y editores hexadecimales).

Hace un par de años, cuando la calidad de los desensambladores no llegaba al nivel de hoy en día, resultaba muy sencillo proteger a los programas del desensamblaje. Se incluían en el código del programa algoritmos y secuencias de instrucciones que el compilador nunca llegaría a generar. De manera que al intentar la descompilación se producía un error o un bucle infinito. He aquí un ejemplo de estas instrucciones "confusas" que se empleaban con frecuencia en el pasado:

```
mov eax,123456h
push eax
ret
```

Este código, de hecho, es equivalente a las instrucciones JMP EAX, y aún se puede hallar de varias formas en numerosos mecanismos de protección.

Hoy en día, los desensambladores están preparados frente a tales tácticas, reduciendo todas las garantías de éxito de esta técnica. Hay que procurar volver inútil la tarea de desensamblar un fichero. Se podrá desensamblar el fichero, pero el código resultante o bien no tendrá ningún sentido, o bien estará tan desordenado que resultará inmanejable para cualquier análisis. Posteriormente se examinará con más detalle esta técnica. A continuación se describirán los mecanismos de protección frente a las propiedades más peligrosas de los desensambladores, por ejemplo, las referencias a las cadenas de caracteres y funciones importadas.

Protección contra las cadenas

Resulta muy sencillo evitar que otros busquen cadenas de caracteres en un programa. Basta con cifrarlos (aparecerán caracteres sin sentido en vez de los esperados, completamente inútiles para un cracker), modificarlos de alguna manera, o preferentemente evitar su uso enteramente si fuera posible. Existen muchas formas de guardar texto en un fichero...

Protección contra las funciones importadas

Los desensambladores emplean una tabla especial, denominada tabla de importaciones, para crear la lista de funciones importadas que apuntan a ciertas ubicaciones dentro del programa. En el capítulo 7, acerca del formato PE, aplicado en la mayoría de los métodos de protección contra funciones importadas, se abundará más en su estructura exacta, funciones y protección. Bastará ahora con detallar las técnicas más simples, basadas enteramente en llamadas indirectas a funciones y en invocaciones hechas de tal forma que a la hora de importarlas, el compilador no las puede tratar de ninguna manera (al intentar insertarlas en la tabla de importaciones). Las funciones invocadas de este modo no se describen en la tabla de importaciones con lo que el desensamblador no podrá mostrarlas en su lista.

Esta técnica resulta útil principalmente con las funciones API, donde en la mayoría de los casos no hay necesidad de cargar la librería que queremos "importar" manualmente de la memoria (mediante la función API `LoadLibraryA`) puesto que normalmente ya la está utilizando el proceso.

El primer paso consiste en localizar la imagen de la librería cargada (véase el capítulo 7), donde se ubica la función solicitada. A continuación se calcula su dirección, y se accede a ella directamente.

En este ejemplo se localiza la dirección de la función `CreateFileA` en la librería kernel32.dl. Es muy simple:

```
HMODULE Handle = GetModuleHandle("kernel32");
                                        // imageBase kernel32
FARPROC Create = GetProcAddress(Handle,"CreateFileA");
                                          // dirección de la
                                       // función CreateFileA
```

A continuación se procede invocando a la función, por ejemplo:

```
__asm
{
        push 0
        push FILE_ATTRIBUTE_NORMAL
        push OPEN_EXISTING
        push 0
        push FILE_SHARE_READ
        push GENERIC_READ
        push pointer_on_file_name
        call Create // CALL CreateFileA
        mov File_Handle,eax
}
// HANDLE File_Handle =
 CreateFile("file_name",GENERIC_READ,FILE_SHARE_READ,
 NULL,OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,NULL);
```

Esta técnica en sí es muy vulnerable por lo fácil que resulta identificar la combinación de las bien conocidas funciones "GetModuleHandleA" y "GetProcAdress" en la tabla de funciones importadas. Es más, aún puede encontrarse en la lista de caracteres los nombres de las funciones y de la librería, lo que no pasará desapercibido. Resulta preferible combinar esta técnica con otras relativas a las funciones importadas y a la protección de cadenas de caracteres.

Este método de protección es mucho más frecuente: la dirección de las funciones importadas se guardan en un sitio (ya sea utilizando la tabla de importaciones o no) al que se invoca desde otro lugar del código. Ésta es una entre tantas maneras de evitar a los desensambladores mostrar la función importada: guardándola en un sitio inhabitual.

En cualquiera de los ejemplos del capítulo 9 se podrá observar el uso de los métodos anteriormente mencionados.

CÓDIGO AUTOMODIFICABLE (SMC)

SMC constituye uno de los métodos de protección de software más utilizados que, además, excede el ámbito del desensamblaje. Puede parecer un poco forzado incluir esta técnica en el capítulo sobre antidesensamblaje, pero téngase en cuenta que muchas técnicas de cracking se complementan y se solapan, volviendo indistinguibles las fronteras entre ellas. Por otra parte, algunas técnicas antidesensamblaje realmente funcionan como antidepuradores, y viceversa. Resulta más cierto aún en el caso de SMC. Obstaculiza la edición del código del programa así como su depurado y desensamblado, incluso lo imposibilita totalmente. Hay dos tipos de SMC: el primero emplea generación (edición) dinámica completa del código del programa durante su ejecución; el segundo saca partido de las múltiples formas de expresar código en ensamblador. Con objeto de simplificar, al

primero se denominará *SMC activo* y al segundo *SMC pasivo*. No obstante, éstos no son términos oficiales.

SMC Pasivo

Obsérvese el siguiente ejemplo:

```
:00401000 EB01 jmp 00401003
:00401002 E86641 call xxxx  El parámetro de esta instrucción no es importante!
:00401005 ...
```

Se habrá observado algo muy extraño: ¡la primera instrucción JMP salta a algún punto en mitad de la siguiente instrucción CALL! se salta un byte en la dirección 00401002. El código realmente queda de la siguiente manera:

```
:00401000 EB01 jmp 00401003
:00401002 E8 // saltada esta instrucción incorrecta.
:00401003 6641 inc eax
```

A partir de este sencillo ejemplo, resulta fácil entender en qué consiste y cómo funciona el SMC pasivo. Con este tipo de instrucción se pueden obtener resultados deslumbrantes. El código queda muy desordenado y dificulta terriblemente su depurado y desensamblado. También entorpece muchísimo la edición del propio código del programa. Resulta muy ardua la tarea de entender la estructura de dicho código.

No existen consejos generales para crear algoritmos SMC. Exige cierta cantidad de tiempo, paciencia y experiencia.

El siguiente ejemplo ilustra este tipo de código:

```
:00401000 EB01 jmps 000401003
:00401002 2853BB sub [ebx][-0045],dl
:00401005 7856 js 00040105D
:00401007 3412 xor al,012
:00401009 EB15 jmps 000401020
:0040100B 2881F3214365 sub [ecx][0654321F3],al
:00401011 87EB xchg ebp,ebx
:00401013 02E8 add ch,al
:00401015 6981F358155195 90EB04 imul eax,d,[ecx][0511558F3],004EB9
:0040101F C2EBEA retn 0EAEB
:00401022 B48B mov ah,08B
:00401024 C3 retn
:00401025 5B pop ebx
```

¿Se podría afirmar que lo único que este código realiza consiste en insertar el valor 1 en el registro EAS? Probablemente no. Examinemos la secuencia de cambios producidos en el código conforme se procesa.

Tras el primer salto a la dirección 00401000, el código se transforma de la siguiente manera:

```
:00401003 push ebx
:00401004 mov ebx,12345678
:00401009 jmp 00401020
:0040100B sub [ecx+654321F3],al
:00401011 xchg ebp, ebx
:00401013 add ch,al
:00401015 imul eax,[ecx+511558F3],04EB9095
:0040101F ret eaeb
:00401022 mov ah,8B
:00401024 ret
:00401025 pop ebx
```

Se ejecutan las instrucciones de las direcciones 00401003 y 00401004 y se salta a 00401020. Tras este salto el código queda así:

```
:00401020 jmp 0040100C
:00401022 mov ah,8B
:00401024 ret
:00401025 pop ebx
```

A continuación se vuelve a la dirección 0040100C:

```
:0040100C xor ebx,87654321
:00401012 jmp 00401016
:00401014 call 59339182
:00401019 adc eax,EB909551
:0040101E add al,c2
:00401020 jmp 0040100C
:00401022 mov ah,8B
:00401024 ret
:00401025 pop ebx
```

Tras procesar las instrucciones en la dirección 0040100C se produce otro salto, esta vez a 00401016:

```
:00401016 xor ebx,95511558
:0040101C nop
:0040101D jmp 00401023
:0040101F ret eaeb
:00401022 mov ah,8B
:00401024 ret
:00401025 pop ebx
```

Y tras el salto final a 0040101D:

```
:00401023 mov eax, ebx
:00401025 pop ebx
```

Si se sobrescribieran todas las instrucciones "reales", se obtendría el código siguiente, a partir del cual no resulta difícil entender cómo funciona el algoritmo:

```
push ebx // saves EBX
mov ebx,12345678h // EBX = 12345678h
xor ebx,87654321h // EBX = 95511559h
xor ebx,95511558h // EBX = 1
mov eax,ebx // EAX = EBX = 1
pop ebx // recupera EBX
```

El código entero equivale a la instrucción `MOV EAX, 1`.

Gracias al uso de SMC y a la utilización de un mayor número de instrucciones para ejecutar una tarea dada (en vez de la sencilla `MOV EAX, 1`), la tarea más simple, como guardar el valor 1 en el registro EAX, llega a convertirse en una operación compleja y muy desordenada. Se hace muy difícil entender la estructura de semejante código. Con algoritmos más largos, y sin un depurador, en ocasiones puede resultar imposible.

El anterior ejemplo no sólo tiene como objeto mostrar el uso práctico del SMC, sino también enseña que no es mala idea en ocasiones escribir algo simple de manera más complicada. Juzgue el lector por sí mismo el resultado y el éxito de ambas técnicas.

SMC Activo

La generación dinámica de código constituye una de las mejores maneras de proteger el código de programa. Ya se recomendó este método al principio del primer capítulo, ahora se describirá y definirá con mayor detalle. Si bien éste no es su lugar idóneo, esta técnica merecería un capítulo íntegro.

Realmente consiste en editar el código del programa en memoria durante su ejecución. Al igual que en ella se escriben variables, registros, etc. también se puede escribir el código del programa. El único problema radica en que el segmento de memoria empleado para escribir debe tener características de escritura, en caso contrario provoca el error "STATUS_ACCSESS_VIOLATION". Asunto relacionado con el formato PE, del cual se encontrará más información en el capítulo pertinente. Bastará con utilizar la función "VirtualProtect" que, con los parámetros oportunos, sí permitirá escribir el código de programa ejecutable.

Con un sencillo ejemplo que ilustre las posibilidades y potencia reales del SMC. Tras echar una ojeada al código siguiente, el lector podrá con seguridad entender su cometido:

```
BOOL A = TRUE;
if (A)
     MessageBox("El código del programa no se ha
      modificado",NULL,MB_OK);
else
     MessageBox("El código del programa se ha modificado
      con éxito",NULL,MB_OK);
```

El bucle causado por "else" resulta lógicamente superfluo: siempre se mostrará el mensaje "El código del programa no se ha modificado". Con ensamblador se puede escribir el código del ejemplo de la siguiente forma:

```
__asm
{
 mov eax,1
 cmp eax,1
 jnz SMC_
}
     MessageBox("El código del programa no se ha
      modificado ",NULL,MB_OK);
     return;
```

```
SMC_:
     MessageBox("El código del programa se ha modificado
      con éxito ",NULL,MB_OK);
```

Seguidamente se puede incluir la llamada a la función "VirtualProtect" en el programa y sustituir la instrucción JNZ en su contraria lógica JZ. Al hacerlo, invertimos la lógica del algoritmo entero para llegar a ejecutar MessageBox con el parámetro "El código del programa se ha modificado con éxito". Éste sería el resultado:

```
DWORD Address,LastProtect;

__asm
{
 mov Address,offset Overwrite
}
VirtualProtect
((void*)Address,10,PAGE_READWRITE,&LastProtect);

__asm
{
 mov ebx,Address
 mov byte ptr [ebx],74h // El valor 75h (instrucción
                        //JNZ) modificado a 74h (JZ)
 mov eax,1
 cmp eax,1
Overwrite:
 jnz SMC_ // esta instrucción quedará sustituida por JZ
}
 MessageBox("El código del programa no se ha
  modificado",NULL,MB_OK);
 return;
SMC_:
 MessageBox("El código del programa se ha modificado con
  éxito",NULL,MB_OK);
```

Tras esta modificación, el mensaje mostrado será: "El código del programa se ha modificado con éxito".

Como puede comprobarse resulta verdaderamente simple. Consiste en un pequeño cambio en el modo de invocar la función "VirtualProtect" para aplicar derechos de escritura a una cierta área de memoria, en nuestro caso, al área donde se cargará el código ejecutable.

Ya se ha dicho anteriormente que éste es un ejemplo sencillísimo; sin embargo, demuestra maravillosamente el uso y potencial del SMC. Dependerá del programador, como siempre, el grado de perfección con el que se aplique esta técnica y el tiempo invertido en el diseño del código del programa.

EDICIÓN DEL CÓDIGO DEL PROGRAMA EN TIEMPO DE EJECUCIÓN

Entre los métodos mejores y más extendidos de protección de los programas frente al cracking se encuentran los que utilizan los principios básicos del SMC activo (por ejemplo, edición del código del programa en tiempo de ejecución). Sus posibilidades son realmente ilimitadas: cifrado del código del programa, modificación de las instrucciones individuales, destrucción intencionada de las instrucciones, y viceversa, inserción y desplazamiento de grandes bloques de datos, sobrescritura de funciones enteras, etc. En pocas palabras: los límites los marca la imaginación del usuario.

Ya se ha puesto de manifiesto en este libro, y se volverá a insistir posteriormente, en las posibilidades de la edición del código de programa en el momento de su ejecución. En el anterior apartado de este capítulo sólo se quería apuntar lo sencillo que resulta. En el capítulo 7 se explicará la utilización del formato PE para cifrar y descifrar datos en tiempo de ejecución.

CAPÍTULO 4

PROTECCIÓN CONTRA FROGSICE

FrogsIce y ProcDump son dos de los programas imprescindibles para un cracker. Representa un antiantidepurador (o más bien un antiantiSoftICE) que puede identificar y engañar a la mayoría de los métodos más recientes para detectar un depurador.

El programa, por explotar las librerías VxD, sólo funciona con sistemas operativos Windows 9x/Me. Esta limitación queda compensada por la gran variedad de funciones sumamente útiles que ofrece el programa: desde identificar las formas básicas de detección de depuradores hasta la supervisión de los vectores de interrupción.

USO ELEMENTAL DE FROGSICE

El corazón de FrogsIce reside en una librería VxD del mismo nombre, que queda cargada en memoria y controlada por un programa denominado FPLoader. Al arrancar, el programa muestra un sencillo menú con diferentes opciones.

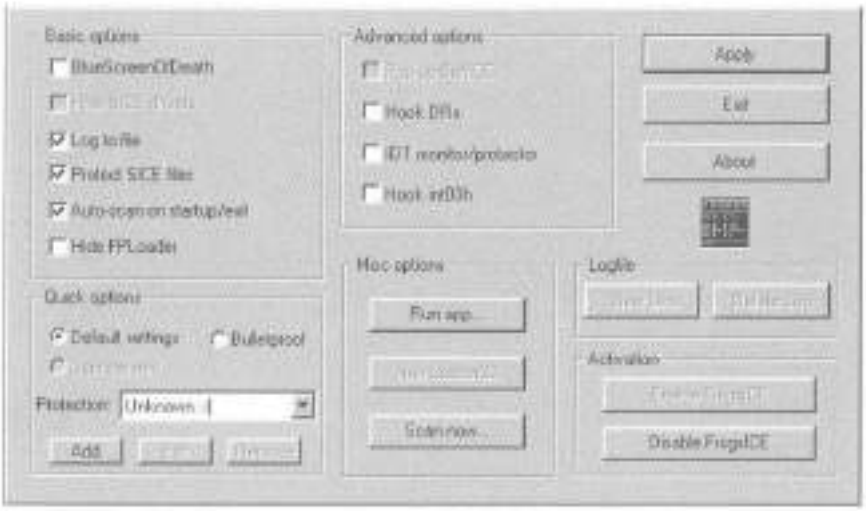

Figura 4-1. Entorno de trabajo de FrogsICE

Al pulsar el botón "`Enable FrogsICE`", el programa carga la librería VxD anteriormente mencionada y quedan activos los servicios del programa. Para desactivarlos, púlsese "`Disable FrogsICE`". A continuación se describen sus opciones.

Opciones básicas

BlueScreenOfDeath – uno de los métodos básicos de alertar al usuario mediante la "muy popular" pantalla azul con todos los detalles sobre la identificación realizada de un algoritmo para detectar depuradores u otra acción semejante.

Hide SICE drivers – oculta los drivers de SoftICE (`SICE`, `SIWDEBUG` y `SIWVID`) de la lista DDB para evitar su detección —por ejemplo, con el método en el que se emplea la función API `CreateFileA`—.

Log to file – todo lo que suceda queda registrado en un fichero; la información se puede recuperar pulsando el botón "View Log".

Protect SICE files – protege los ficheros de SoftICE contra su eliminación o modificación. Algunos autores de algoritmos antidepurador ponen tanto empeño en su misión que tras detectar un depurador, lo borran. Con lo que corren el riesgo de ser demandados judicialmente por un cracker.

Auto-scan on startup/exit – con cada arranque o parada de FrogsICE se realizan comprobaciones de áreas importantes de memoria (como IDT) buscando alguna modificación u otras características "sospechosas". Si se desea realizar una comprobación al instante, púlsese el botón "Scan now".

Hide FPLoader – oculta el fichero de arranque de FrogsICE.

Beep! – se puede elegir como alerta un tono normal en vez de la pantalla BlueScreenOf Death.

Opciones avanzadas

Pop-up SoftICE – cuando una de las opciones se detecte, se mostrará un mensaje en una dirección específica indicando que SoftICE está listo.

Hook DRx – esta opción ayuda a supervisar el acceso (en escritura o lectura) a los registros de depuración (Dr0 – Dr7). Ha de tenerse mucho cuidado puesto que puede provocar la caída del sistema. Se recomienda desactivarla o borrar todos los puntos de corte hardware definidos en SoftICE antes de activarla. Consúltese la documentación para asegurar su utilización correcta.

IDT monitor/protector – garantiza la detección de cualquier intento de utilizar IDT (lista con los descriptores de las interrupciones). Herramienta sumamente útil para buscar protecciones mediante vectores de interrupción (sustituciones de vector, bifurcación al anillo 0, etc.).

Hook int03h – en caso de que no esté activo SoftICE, esta opción puede detectar cualquier llamada a INT3. Muy útil, puesto que la interrupción se utiliza a menudo en algoritmos de protección de todo tipo.

Adjust localtime to RTC – esta opción puede engañar a algunos algoritmos antidepurador basados en la medición del tiempo de proceso de algunas partes importantes del código de programa.

ALGORITMOS COMUNES

VxDCall de la función VMM_GetDDBList

Algunos algoritmos dedicados a evitar la utilización de Frogslce utilizan la VxDCall de la función `VMM_GetDDBList`, en cuyo caso, si FrogsICE estuviera activo, causaría una excepción (la famosa pantalla azul). Al igual que se mencionó anteriormente con el uso de VxDCall, esta función debe invocarse desde el anillo 0. Así queda su uso limitado a Windows 9x/Me, lo que carece de importancia puesto que FrogsICE no se puede utilizar con otros sistemas. Según se observa en los ejemplos de las secciones anteriores, el empleo de este método no debe suponer ningún problema. Basta con mover el programa al anillo 0 y ejecutar la función `VMM_GetDDBList`. Si FrogsICE estuviese activo, el programa generaría una excepción y, probablemente finalizaría. Si FrogsICE no estuviese activo, no

sucederá nada y el programa proseguirá su ejecución. Un ejemplo de dicho código puede ser el siguiente:

```
__asm
{
    ...
    push edx
    sidt [esp-2]              // lectura de IDT en la pila
    pop edx

    add edx,(3/*=INT 3*/*8)+4       // lectura del vector
                                       de interrupciones
                                       seleccionado, esto
                                       es, INT 3
    mov ebx,[edx]
    mov bx,word ptr [edx-4]         // guardando la
                                       dirección de la
                                       anterior rutina de
                                       interrupciones
    lea edi,Ring0
    mov [edx-4],di
    ror edi,16                      // definición de una
                                       rutina de
                                       interrupción INT3
                                       nueva
    mov [edx+2],di

    push ds
    push es
    INT 3                           //-->anillo 0
    pop es
    pop ds
    mov [edx-4],bx                  // recuperación de la
                                       rutina de
                                       interrupción INT3
                                       anterior
    ror ebx,16
    mov [edx+2],bx
    jmp Ok


/******* anillo 0*******/
Ring0:
    __emit 0xCD                     // VMMCall
    VMM_GetDDBList
    __emit 0x20                     //
```

```
    __emit 0x3F                    //
    __emit 1                       //
    __emit 1                       //
    __emit 0                       //
    iretd                          // si se ha llegado
                                      hasta aquí, FrogsICE
                                      no reside en memoria
Ok:
    ...

    }
```

Este ejemplo constituye una demostración ideal de lo falso del supuesto que afirma lo muy difícil que resulta utilizar la sustitución de vector para saltar al anillo 0. Afirmación ya realizada varias veces anteriormente. Basta con activar la opción "IDT monitor/protector" de FrogsICE. Esta opción garantiza la detección de cualquier intento de utilizar IDT e informa al usuario de ello. Además, no se llegará a invocar la función VMM_GetDDBList, lo que volverá inútil al algoritmo. Si se desea saltar al anillo 0, deberá utilizarse otro método que no pueda ser detectado por FrogsICE. Parece que la mejor alternativa reside en SEH. En el ejemplo siguiente se provoca la excepción C0000005h –STATUS_ACCESS_VIOLATION para dar paso a nuestro manejador:

```
    __asm
{
    mov ax,ds
    test al,4                      // ¿sistema operativo?
    je Not_Win9x                   // bifurcación = el
                                      sistema operativo no
                                      es Windows 9x/Me

Win9x:
    push offset MyHandler          // guardando la
                                      dirección de nuestro
                                      manejador
    push dword ptr fs:[0]          // guardando la
                                      dirección del
                                      manejador anterior
    mov dword ptr fs:[0],esp       // instalación
    pushfd
    mov eax,esp
    xor ebx,ebx                    // EBX = 0
  mov ecx,[ebx]                    //--> anillo 0
                                     (STATUS_ACCESS_VIOLATION)
    /********Ring0********/
    push edx //GS
    push edx //FS
```

```
    push edx //ES
    push edx //DS
    push edx //SS
    push eax //ESP
    push dword ptr [eax]              // EFLAGS
    push ecx                          // CS
    push offset Ring3                 // EIP - dirección
                                             para volver
                                             al anillo 3

    __emit 0xCD                       // VMMCall
                                             VMM_GetDDBLi
                                             st
    __emit 0x20                       //
    __emit 0x3F                       //
    __emit 1                          //
    __emit 1                          //
    __emit 0                          //
    iretd                             //--> anillo 3
                                      // si se ha llegado
                                         hasta aquí, FrogsICE
                                         no reside en memoria

Ring3:
    popfd
    pop dword ptr fs:[0]              // recuperación del
                                         manejador
                                         anterior
    add esp,4                         // ajuste de pila
    jmp Ok

    /*******MyHandler*******/
    // SEH handler for jump to Ring0
MyHandler:
    mov edx,[esp+0Ch]                 // CONTEXT
    mov ecx,[esp+4]                   // EXCEPTION_RECORD
    mov ecx,[ecx]                     // ECX = valor de la
                                         excepción recién
                                         ocurrida
    cmp ecx,0C0000005h                // ¿excepción
                                         STATUS_ACCESS_VIOLATION
                                         (C0000005h)?
    jne Error                         // bifurcación =
                                         excepción que no se
                                         procesará por este
                                         manejador
```

```
    add dword ptr [edx+0B8h],2          // recuperación de la
                                           excepción saltando
                                           la instrucción
                                           incorrecta

    movzx ecx,word ptr [edx+0BCh]
    mov [edx+0ACh],ecx
    mov dword ptr [edx+0BCh],28h
    movzx ecx,word ptr [edx+0C8h]
    mov [edx+0A8h],ecx
    mov dword ptr [edx+0C8h],30h
    or dword ptr [edx+0C0h],200h
    sub eax,eax                         //ExceptionContinueExecution
    ret                                 //--> anillo 0

Error:
    sub eax,eax
     inc eax                            //ExceptionContinueSearch
    ret
}
Not_Win9x:
   MessageBox("Este método sólo funciona con Windows
    9x/Me",NULL,MB_OK);
    return;
Ok:
    MessageBox("FrogsICE no detectado",NULL,MB_OK);
```

Uso de la función CreateFileA

Algunas de las versiones antiguas de FrogsICE podían detectarse, al igual que SoftICE, identificando sus drivers mediante la función `CreateFileA` (o `CreateFileW`). En la práctica resulta el método idéntico al empleado para la detección de SoftICE, únicamente varía el nombre del driver.

```
HANDLE File = CreateFile("\\\\.\\FROGSICE",GENERIC_READ |
 GENERIC_WRITE, FILE_SHARE_READ |
 FILE_SHARE_WRITE,NULL,OPEN_EXISTING,
 FILE_ATTRIBUTE_NORMAL, NULL);

if(File != INVALID_HANDLE_VALUE)        // ¿driver de FrogsICE
                                        detectado?
```

```
{
    CloseHandle(File);
    MessageBox("FrogsICE detectado",NULL,MB_OK);
}
else
    MessageBox("FrogsICE no detectado",NULL,MB_OK);
```

Sin embargo, este método no funciona con las versiones más recientes de FrogsICE. FrogsICE informa de todo, y por razones de seguridad y para evitar una excepción potencial con colapso del sistema incluido, elude cualquier intento de detección de este tipo.

CAPÍTULO 5

PROTECCIÓN CONTRA PROCDUMP

ProcDump es un descodificador-descompresor PE universal, además realiza volcados de memoria, editor y regenerador de ficheros PE. Constituye un complejo programa que reúne todo lo necesario para descodificar y descomprimir ficheros PE.

Este programa, gracias a su increíble flexibilidad, y a pesar de que su desarrollo se paralizó hace casi dos años, tiene la posibilidad de ampliar constantemente sus características y propiedades. Con cierta regularidad aparecen plug-ins en Internet para su utilización en el mercado de protección de software, en expansión constante.

USO ELEMENTAL DE PROCDUMP

La interfaz del programa, nada más arrancar, suele provocar una sonrisa de complacencia en el usuario.

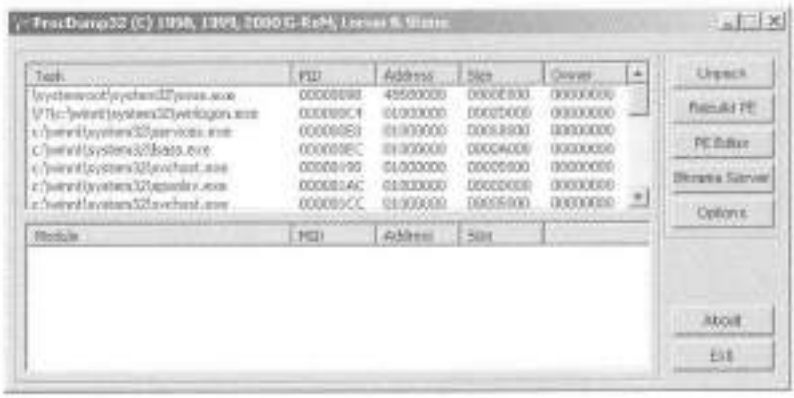

Figura 5-1. Menú básico de ProcDump

Su utilización resulta muy sencilla.

Al pulsar el botón "Unpack", el programa muestra una lista de todos los codificadores y compresores PE que ProcDump puede tratar automáticamente. Se puede elegir entre las posibilidades siguientes (versión 1.6.2):

- Aspack<108
- Aspack108
- Aspack108.2
- Aspack108.3
- Aspack108.4
- Aspack2000
- CodeSafe 3.X
- Hasiuk/NeoLite
- Manolo
- Neolite2
- PCGUARD v2.10

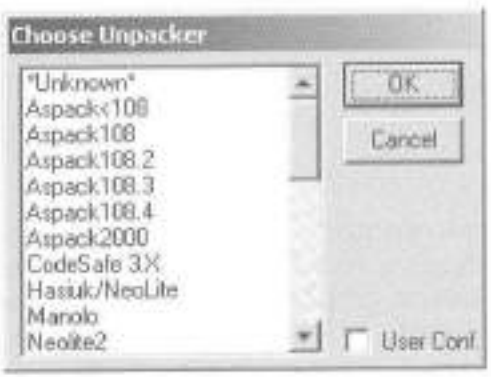

Figura 5-2. Diferentes alternativas de ProcDump

- PCShrink
- PCShrink II
- PE-Compact
- PE-Pack
- PESHiELD
- Petite 2.0
- Petite<1.3
- PKLiTE
- Sentinel
- Shrinker 3.2
- Shrinker 3.x
- SoftSentry
- Standard
- UPX

- Vbox Dialog
- Vbox Std
- VGCrypt 0.75
- Wwpack32

El procedimiento que rige el uso de los anteriores compresores PE se describe claramente con el lenguaje de programación aplicable al fichero script.ini. No constituye, por tanto, ningún problema examinar los métodos individuales de protección y, más importante aún, modificarlos y añadir otros codificadores completamente nuevos que ProcDump pueda utilizar. Todo queda escrito y explicado en el fichero script.txt.

A continuación, se elige un fichero y un método de protección, ProcDump realizará el resto. Si todo va bien, sólo quedará seleccionar el nombre del fichero resultante.

Éste, en ocasiones, exige reconstruirse de una forma concreta. Para ello utilícese el botón "Rebuild PE". En la ventana "Options" se pueden elegir diferentes posibilidades para reconstruir el fichero.

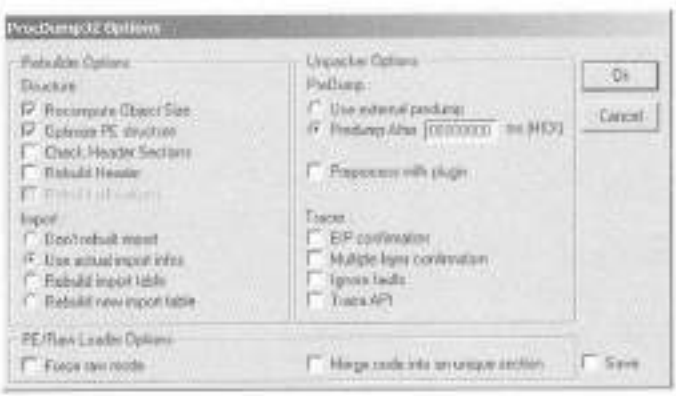

Figura 5-3. Distintas posibilidades de configuración en ProcDump

Generalmente se suele reconstruir la tabla de importaciones (si fuera posible), modificar la cabecera del fichero PE, y otras acciones similares. Estas operaciones varían según el método protección.

El botón "PE Editor" muestra las distintas posibilidades del editor de ficheros PE.

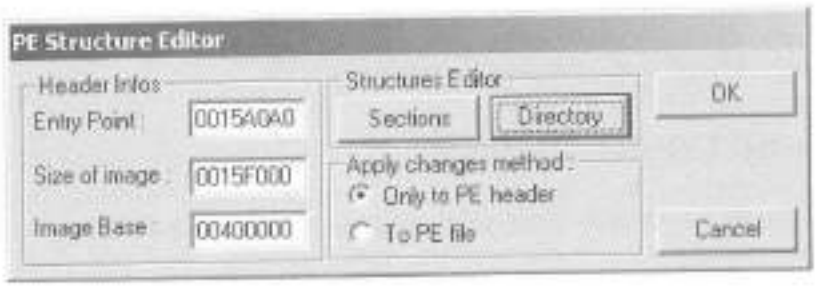

Figura 5-4. Editor de ficheros PE de ProcDump

Resulta, sin embargo, preferible emplear un editor PE especializado como el incluido en el CD adjunto.

El botón "Bhrama Server" activa la aplicación servidor encargada de suprimir enteramente otro tipo de protecciones, como Securom, VBox y Petite. Cada una de ellas exige su cliente.

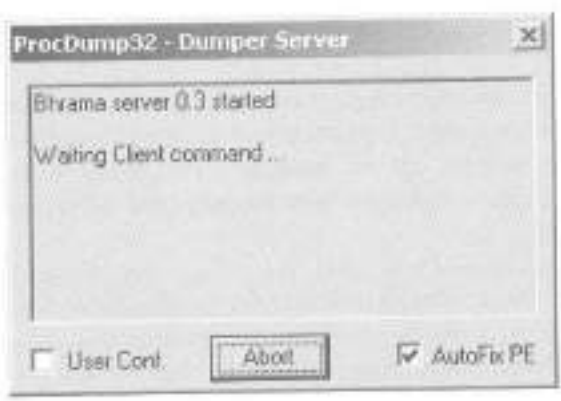

Figura 5-5. Brahma Server a la espera de respuesta del cliente

En el capítulo 7 se examinarán con más detalle éstas y otras opciones aplicables a los ficheros PE. En el capítulo 9, dedicado a ejemplos de cracking, se describirán las restantes funciones de ProcDump, como el volcado de memoria.

DEFINICIÓN Y OBJETIVO DEL VOLCADO DE MEMORIA

La técnica de volcar el contenido de la memoria al disco se utiliza principalmente cuando un programa descifra (o modifica de alguna manera) ciertos datos en tiempo de ejecución porque la lógica del programa exija esos datos descifrados. A un cracker le bastaría esperar a que los datos se descifren para guardarlos y examinarlos posteriormente.

Los volcados de memoria constituyen el punto de partida para eliminar las protecciones basadas en compresores-codificadores PE.

ALGORITMOS COMUNES

Una de las acciones elementales que cualquier algoritmo de protección debe incluir consiste en evitar un volcado realizado por ProcDump. Más aún en el caso de los compresores-codificadores PE, donde el volcado de memoria constituye un punto esencial en la supresión de protecciones. Resulta evidente que con ProcDump se puede descomprimir la inmensa mayoría de codificadores y compresores PE de hoy en día, lo que revela el hecho de que estos algoritmos apenas ya se utilizan para proteger software.

Resulta pueril pensar que por utilizar un algoritmo que imposibilite el volcado con ProcDump, un programa va a quedar protegido contra el volcado de memoria o contra la supresión del compresor-codificador PE. Y más infantil resulta todavía cuando un cracker puede volcar la memoria y descifrar manualmente un fichero mediante uno de los muchos otros programas equivalentes disponibles en Internet, o bien utilizando un descompresor distinto (y, por supuesto, suprimiendo dicho algoritmo).

Por otra parte, en cierta medida este algoritmo protege de forma general contra la supresión de los mecanismos de protección y contra el volcado realizado por uno de los programas más extendidos con tal propósito: ProcDump. Como mínimo, el programa estará defendido frente a todos los principiantes, que se verán obligados a buscar una alternativa diferente.

Se puede evitar el volcado realizado por ProcDump modificando la cabecera PE del fichero almacenado en memoria y utilizado internamente por el sistema. Si el tamaño de un proceso dado cambia a 1, ProcDump colapsará al intentar volcar el proceso. Dicho tamaño puede también aumentarse, disminuirse o modificarse de alguna forma, pero en la mayoría de los casos ProcDump logrará, al menos parcialmente, volcar tal proceso (pruebe el lector y lo comprobará).

El proceso para modificar el tamaño del fichero en su cabecera PE con Windows NT/2000/XP difiere al de Windows 9x/Me. Por tanto, resulta imprescindible comprobar la plataforma al principio del algoritmo.

```
__asm
{
    push fs:[30h]
    pop eax
    test eax,eax        // ¿sistema operativo ?
    js Win9x            // bifurcación - el sistema
                           operativo es Windows 9x/Me
```

```
Not_Win9x:
      mov eax,[eax+0Ch]
      mov eax,[eax+0Ch]
      mov dword ptr [eax+20h],1 // ajuste de tamaño
      jmp End
}

Win9x:
     GetModuleHandle(NULL);       // con Windows 9x/Me
                                     esta  función copia
                                     en  EDX los datos
                                     solicitados
__asm
{
      test edx,edx
      jns End // ¿se invocó la función con éxito?
      cmp dword ptr [edx+8],-1
      jne End // segunda comprobación
      mov edx,[edx+4]
      mov dword ptr [edx+50h],1 // ajuste de tamaño
}
End:
MessageBox("Inténtese ahora volcar este proceso con
 ProcDump",NULL,MB_OK);
```

CAPÍTULO 6

EDICIÓN DEL CÓDIGO DEL PROGRAMA

Probablemente sea superfluo destacar la importancia que tiene la edición del código del programa en el cracking. En sentido estricto, se podría afirmar que la modificación del código original del programa constituye una actividad ilegal en sí misma. Al menos, ésa suele ser la opinión de los profanos.

En muchas ocasiones, el cracker puede alcanzar su objetivo sin editar el código del programa (obteniendo el número de serie correcto o la contraseña, reconstruyendo el fichero clave, etc.). En otras, no queda más remedio.

Generalmente el término "edición" se relaciona con un editor: es decir, un programa que facilita la edición. En este contexto, resulta preferible emplear un editor que sea capaz de trabajar con el código máquina del programa de manera legible. La edición del código máquina en formato binario puede llegar a ser inmanejable. Razón por la que se utiliza el formato hexadecimal. De donde procede el nombre del propio editor: editor hexadecimal. Algunos de ellos son capaces inclusive de mostrar directamente el código máquina en ensamblador, al igual que lo hace un desensamblador (seguidamente se examinará uno de estos editores).

A menudo se emplea el término "parchear" para referirse a la edición del código del programa, y por extensión, a los cracks que modifican el código de programa: parches.

MÉTODOS PARA EDITAR EL CÓDIGO DEL PROGRAMA

La edición del código de programa mediante un editor hexadecimal no constituye en absoluto el único medio para modificar el código del programa. Además de editar directamente un fichero dado en el disco duro mediante un editor hexadecimal, también se puede editar el fichero una vez que se cargue en memoria. A diferencia del editor hexadecimal, donde las modificaciones se realizan directamente en un fichero almacenado en el disco duro, con el segundo método es la ubicación (o ubicaciones) de memoria donde resida el fichero lo que se modifica (normalmente mediante la función API `WriteProcessMemory`) y no el fichero propiamente dicho.

En las siguientes secciones de este capítulo se muestran las principales diferencias entre estas dos técnicas y la forma de defenderse ante ellas.

USO ELEMENTAL DE HIEW

Se puede elegir entre la amplia variedad de editores disponibles aquel que más convenga. Hiew constituye uno de los mejores actualmente.

Su apariencia, típica de los sistemas DOS, no debiera desanimar al posible usuario. Sus propiedades superan claramente a las de la mayoría de los editores hexadecimales, al empezar a utilizarlo enseguida se aprecia su facilidad de uso y rapidez.

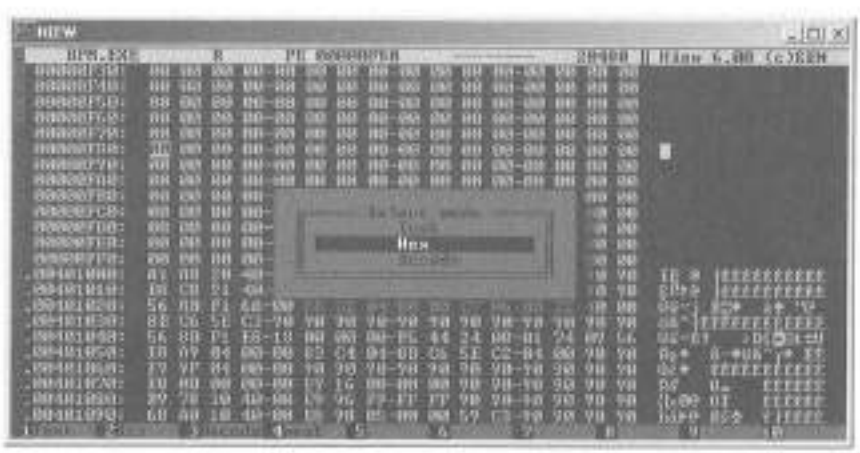

Figura 6-1. Aspecto de Hiew, típico de DOS

Pulsando la tecla F4 se ofrecerán diferentes modos de presentación. Hiew muestra el fichero en edición de tres modos diferentes: en modo texto ("Text"), en formato hexadecimal con texto ("Hex"), o directamente las instrucciones en ensamblador ("Decode"). Se puede saltar de un modo representación a otro pulsando la tecla intro.

Desde el modo hexadecimal o ensamblador, al pulsar la tecla F5 el usuario podrá desplazarse a cualquier ubicación del fichero indicando su desplazamiento real ("raw offset", en inglés). La dirección debe introducirse en formato hexadecimal.

La tecla F3, desde el modo hexadecimal o ensamblador, sirve para editar el fichero. Desde este último modo se puede introducir directamente una instrucción en ensamblador pulsando la tecla de F2. Muy útil cuando no se sabe la expresión numérica de las instrucciones individuales (en formato hexadecimal en nuestro caso). El resultado de la petición se guarda pulsando la tecla F9.

Otra característica muy útil consiste en la capacidad de búsqueda tanto en formato ASCII como en código hexadecimal. La tecla F7 muestra el cuadro de diálogo pertinente.

Hasta aquí una breve descripción de las características básicas del programa Hiew. Seguidamente se describirán otras características muy atractivas.

Edición de un programa para detectar SoftICE

Se examinará ahora cómo funciona Hiew en la práctica. Se tomará como ejemplo el código del programa empleado en el capítulo 2. El código que utiliza la función API `CreateFileA` quedará modificado para que detecte el programa SoftICE y que, independientemente de resultado de la detección, genere un mensaje indicando que no se detectó a SoftICE. A continuación se presenta el código de ejemplo:

```
HANDLE File;
BYTE Win9x = 0;
__asm
{
      push fs:[30h]
      pop eax
      test eax,eax // ¿sistema operativo?
      jns Not_Win9x // bifurcación = el sistema
                    operativo es Windows 9x/Me
      mov byte ptr Win9x,1

 Not_Win9x:
}
if (Win9x)
{
      File = CreateFile( "\\\\.\\SICE",GENERIC_READ |
       GENERIC_WRITE,FILE_SHARE_READ |
       FILE_SHARE_WRITE,NULL,OPEN_EXISTING,
       FILE_ATTRIBUTE_NORMAL,NULL);
}
```

```
else
{
      File = CreateFile("\\\\.\\NTICE",GENERIC_READ |
       GENERIC_WRITE,FILE_SHARE_READ |
       FILE_SHARE_WRITE,NULL,OPEN_EXISTING,
       FILE_ATTRIBUTE_NORMAL,NULL);
}
if( File != INVALID_HANDLE_VALUE ) // ¿driver de
                                   SoftICE detectado?
{
      CloseHandle(File);
      MessageBox("SoftICE detectado",NULL,MB_OK);
}
else
      MessageBox("SoftICE no detectado",NULL,MB_OK);
```

En un programa de estas dimensiones resulta sencillo detectar el algoritmo mediante un editor hexadecimal. Ahora bien, puesto que no se suele utilizar un editor hexadecimal para recorrer un fichero tan pequeño en búsqueda del algoritmo, seguidamente se describirá cómo localizar el objetivo deseado con el desensamblador para luego utilizar la dirección obtenida con Hiew.

Ábrase en primer lugar el fichero con el desensamblador W32Dasm. A continuación ha de diseñarse una estrategia para localizar el algoritmo de detección de SoftICE. Esta ubicación se puede hallar bien aplicando el código dado donde figura las sentencias "SoftICE detectado" o "SoftICE no detectado", o bien utilizando las funciones importadas —función API `CreateFileA`—. Selecciónese la opción preferida (doble pulsación en la función/mensaje seleccionado). El código siguiente muestra el programa en ensamblador:

```
Possible StringData Ref from Data Obj ->"\\.\SICE"

:00401364 6850304000 push 00403050
:00401369 EB05 jmp 00401370

* Referenced by a (U)nconditional or (C)onditional Jump
* at Address: 00401362(C)

* Possible StringData Ref from Data Obj ->"\\.\NTICE"

:0040136B 6844304000 push 00403044

* Referenced by a (U)nconditional or (C)onditional Jump
* at Address: 00401369(U)
```

```
* Reference To: KERNEL32.CreateFileA, Ord:0034h

:00401370 FF1500204000 Call dword ptr [00402000]
:00401376 83F8FF cmp eax,FFFFFFFF <-- if (File !=
                                  <-- INVALID_HANDLE_VALUE)
 :00401379 741C je 00401397 <-- bifurcación = SoftICE no
                                              <-- detectado

:0040137B 50 push eax

* Reference To: KERNEL32.CloseHandle, Ord:001Bh

:0040137C FF1508204000 Call dword ptr [00402008]
:00401382 6A00 push 00000000
:00401384 6A00 push 00000000

* Possible StringData de Data Obj ->"SoftICE detectado"

:00401386 6834304000 push 00403034
:0040138B 8BCE mov ecx,esi

* Reference To: MFC42.Ordinal:1080, Ord:1080h

:0040138D E81E020000 Call 004015B0
:00401392 5E pop esi
:00401393 8BE5 mov esp,ebp
:00401395 5D pop ebp
:00401396 C3 ret

* Referenced by a (U)nconditional or (C)onditional Jump
* at Address:00401379(C)

:00401397 6A00 push 00000000
:00401399 6A00 push 00000000

* Possible StringData de Data Obj ->"SoftICE not found"

:0040139B 6820304000 push 00403020
:004013A0 8BCE mov ecx, esi

* Reference To: MFC42.Ordinal:1080, Ord:1080h

:004013A2 E809020000 Call 004015B0
:004013A7 5E pop esi
:004013A8 8BE5 mov esp,ebp
:004013AA 5D pop ebp
:004013AB C3 ret
```

El código resulta muy simple. Si se produce una bifurcación en la dirección 00401379, se mostrará el mensaje "SoftICE no detectado". Por tanto, si se modifica la bifurcación para que en vez de condicional sea incondicional (se produce siempre), el mensaje se mostrará siempre. (Por supuesto ésta no es la única solución posible.)

Defínase el indicador de la instrucción de bifurcación condicional JE a la dirección 00401379 y obsérvese la barra de estado de W32Dasm. Se apreciará el valor de desplazamiento 00001379h, que define su posición real en el fichero. De esta manera se dispone de toda la información necesaria para editar el fichero.

Ábrase Hiew con el programa que se está estudiando. Pulse dos veces la tecla intro para operar en modo ensamblador. Seguidamente F5 y el valor de desplazamiento (1379h, sin la "h" por supuesto). La herramienta se situará en la ubicación de la instrucción de bifurcación condicional (JE). Púlsese F3 para editar la edición y modificar el valor inicial de la instrucción de bifurcación 74h a EBh. De este modo, la instrucción de bifurcación condicional se convierte en incondicional (JMP). Sólo resta pulsar F9 para guardar las modificaciones en el fichero. Ya está. El programa siempre indicará que SoftICE no se ha detectado, independientemente del resultado de la detección.

Con este ejemplo se pretendía ilustrar no sólo cómo utilizar el editor hexadecimal Hiew, sino también lo fácil que resulta someter una aplicación no protegida con un desensamblador y un editor hexadecimal. En la siguiente sección se describirá cómo protegerse frente a esta técnica.

ALGORITMOS COMUNES

Comprobación de la integridad de los datos

Probablemente sea la comprobación de la integridad de los datos la manera más conocida de proteger el código de un programa frente a su edición. Al igual que se comprueba la integridad de los paquetes de datos enviados a través de Internet, una aplicación también puede comprobar su propia integridad.

Como con todo lo demás, el éxito de este tipo de protección dependerá de su correcta aplicación y del modo en que el algoritmo de protección reaccione cuando detecte intentos de vulnerarlo.

COMPROBACIÓN DE LA INTEGRIDAD DE LOS DATOS DE UN FICHERO

Los primeros algoritmos para comprobar la integridad de los datos empleaban las funciones API `CreateFileA` y `ReadFile`, con ellas se cargaba en memoria el contenido de un fichero para luego efectuar la comprobación mediante el algoritmo correspondiente.

El código podría asemejarse al siguiente:

```
DWORD NOBR;
HANDLE File =
 CreateFile("file_name",GENERIC_READ,FILE_SHARE_READ,
 NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
                        // obtención del manejador de fichero
BYTE *pMem = new BYTE[GetFileSize(File,NULL)];
                                              // asignar memoria
ReadFile(File,pMem,GetFileSize(File,NULL),&NOBR,NULL);
                                // cargar fichero en memoria
DWORD CheckSum =
CalculateCheckSum(pMem,checked_data_area...etc.);
if (CheckSum == CorrectCheckSum)
{
   // todo correcto...
}
...
CloseHandle(File);

    delete pMem;
```

Se puede encontrar un ejemplo práctico de este algoritmo en el CD incluido en este libro y también en algunos programas del capítulo 9.

También se suele aplicar la función `MapFileAndCheckSum` con el mismo objetivo. En este caso, desgraciadamente, resulta imposible definir el área de datos procesada y ha de efectuarse necesariamente la comprobación de integridad con el fichero entero. Más aún, dicha función sólo se puede aplicar a ficheros compatibles con el formato PE (véase el siguiente capítulo).

Esta función no resulta muy apropiada si el programa ha de comprobar su propia integridad ya que siempre será obligatorio guardar el valor correcto de la comprobación en una ubicación que quede fuera de su cálculo, lo que no resulta tan fácil. A diferencia del algoritmo anterior, no es posible controlar el área de datos contra los que se efectúa la comprobación. El área donde se guarde el valor correcto de la comprobación podrá ser bien la variable `CheckSum`, en la estructura `IMAGE_OPTIONAL_HEADER32` de la cabecera del fichero con formato PE, donde "nadie" se lo esperaría, o en una ubicación fuera de fichero, lo que podría resultar demasiado lento y no suficientemente seguro.

Siempre que se utilice esta función resultará lógicamente imposible guardar directamente en el código del programa el valor correcto de la comprobación —el programa deberá contener el valor de la comprobación calculada, que sólo se podrá hallar después de que se ejecute el algoritmo que la calcula—, puesto que este algoritmo también

abarca la parte del código que contiene la verificación del valor calculado de la comprobación.

La función `MapFileAndCheckSum` constituye una solución excelente para realizar comprobaciones de integridad de otros ficheros tipo PE, como librerías DDL, con rapidez y comodidad. Su principal ventaja radica en que el resultado de la función `MapFileAndCheckSum` es el valor calculado de la comprobación. Lo que evita codificar un algoritmo diferente para calcularlo, a diferencia del ejemplo anterior.

El código resulta muy sencillo:

```
DWORD PECheckSum,CalculatedCheckSum;
MapFileAndCheckSum("file_name",&PECheckSum,&CalculatedChec
 kSum);
```

La función obtiene dos valores de comprobación. El primero, guardado en la variable `PECheckSum`, recoge los contenidos de la variable anteriormente mencionada `CheckSum` de la estructura `IMAGE_OPTIONAL_HEADER32` de la cabecera del fichero PE. Según se mencionó anteriormente, se puede utilizar esta variable para verificar la segunda comprobación, guardada en la variable `CalculatedCheckSum`. Esta segunda variable representa el valor correcto de la comprobación efectuada sobre un fichero dado. En el CD incluido en este libro se puede encontrar un ejemplo práctico de este algoritmo.

El primer punto débil, obvio por otra parte, de estos algoritmos radica en el uso de funciones muy conocidas. Inmediatamente quedó de manifiesto que éste no era el único problema con este tipo de protecciones. Su principal deficiencia consiste en que la comprobación se realiza sobre el contenido del fichero en disco y no sobre el contenido en memoria, donde el programa queda cargado al ejecutarse. Los crackers, en vez de editar el fichero directamente desde el disco con un editor hexadecimal, lo editan desde la memoria con el programa ya cargado, con lo que la protección quedará anulada. Los programas destinados a esta tarea se denominan cargadores. Con frecuencia ejecutan el programa con la función API `CreateProcess` para luego aplicar los cambios en memoria mediante la función `WriteProcessMemory`.

Estos métodos para comprobar la integridad de los datos apenas se utilizan ya, o mejor dicho, no deberían utilizarse. Como la excepción confirma la regla, no sería extraño encontrar algunos sistemas de protección modernos que apliquen este método.

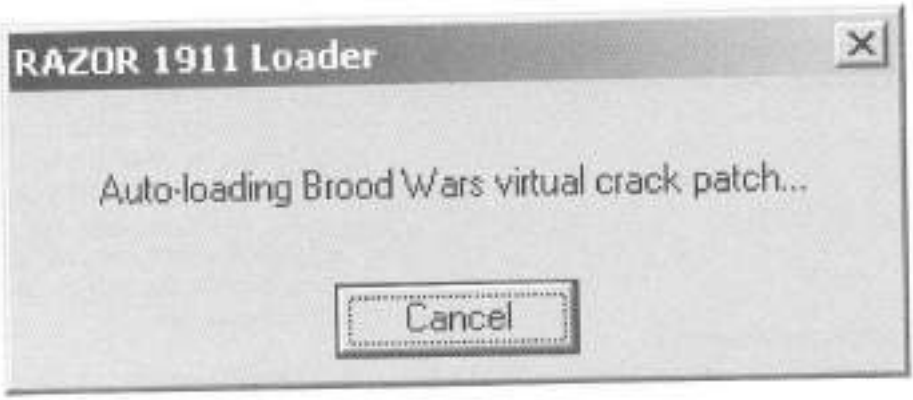

Figura 6-2. Cargador StarCraft

Los cargadores también se suelen utilizar para editar programas protegidos con compresores-codificadores PE. En este caso, el cargador espera hasta que el fichero se descifre y los datos solicitados se carguen en el área de memoria que contiene el programa descifrado. No se profundizará ahora en este tema puesto que el capítulo siguiente está dedicado por completo al formato PE.

Además de un generador de cargadores, en el capítulo 8 se describirá cómo programar cargadores. A continuación se muestra un código ejemplo de cargador:

```
DWORD OverwriteAddress,NOBW;
BYTE OverwriteValue;
PROCESS_INFORMATION ProcessInfo;
STARTUPINFO StartInfo;
StartInfo.cb = sizeof (StartInfo);
StartInfo.lpReserved = NULL;
StartInfo.lpDesktop = NULL;
StartInfo.lpTitle = NULL;
StartInfo.dwFlags = 0;
StartInfo.cbReserved2 = 0;
StartInfo.lpReserved2 = NULL;
OverwriteAddress = 0x004014B6; // dirección de memoria
                               // donde se aplican las
                              //modificaciones, por
                             //ejemplo 004014B6 se
                             // escribe:
OverwriteValue = 0x74        //en una ubicación dada de
                             //memoria, por ejemplo 74h
```

```
CreateProcess
 ("file_name",NULL,NULL,NULL,false,CREATE_SUSPENDED,
 NULL,NULL,&StartInfo,&ProcessInfo); // crea el proceso de
                                     // la aplicación dada
                                     //con el parámetro
                                      //CREATE_SUSPENDED,
                                      // con lo que el
                                    // programa se cargará
                                   //en memoria pero no se
                                    //ejecutará
WriteProcessMemory
 (ProcessInfo.hProcess,(LPVOID)OverwriteAddress,
 &OverwriteValue,sizeof (OverwriteValue),&NOBW);
                              // aplica los cambios en memoria
    ResumeThread (ProcessInfo.hThread);
                                     // ejecuta la aplicación
```

COMPROBACIÓN DE LA INTEGRIDAD DE LOS DATOS EN MEMORIA

Resulta preferible efectuar la comprobación de integridad directamente en memoria una vez que el programa se haya cargado en su arranque. Además de que, a diferencia del método anterior, esta técnica no exija cargar el fichero (o partes de él) repetidas veces en memoria (donde ya reside), sortea los principales problemas de seguridad (uso de funciones API bien conocidas, etc.) que tal método comportaba. Resulta también mucho más simple y por añadidura, permite detectar cualquier punto de corte bxp definido. Estos puntos de corte sobreescriben los datos al valor CCh, lo que provoca un resultado incorrecto en la comprobación de integridad.

Resulta importante, sin embargo, destacar la siguiente observación. Al aplicar los antiguos mecanismos para comprobar la integridad de los datos en disco, no tiene ninguna importancia cuándo y cuántas veces se realice la comprobación, puesto que el cargador modifica el programa en memoria. Al efectuar la comprobación de integridad en memoria, por contra, el programador debe recordar cuándo y cuántas veces se calcula. Si sólo se realizase una vez en el arranque, la protección será muy vulnerable frente a los cargadores. Al igual que sucede con los compresores-codificadores tipo PE, al cargador le bastará con esperar a que el fichero quede descomprimido para realizar sus modificaciones, en este caso habrá de esperar a que finalice la comprobación de integridad para modificar los contenidos en memoria. Se hace necesario, por tanto, efectuar comprobaciones de integridad constante y repetidamente, si fuera posible además, enteramente al azar. Bajo ninguna circunstancia, el programador debiera definir una sola función a la que se invoque repetidas veces, puesto que la mayoría de las ocasiones bastaría con anular esta sola función. Al igual que con muchos otros tipos de protección (comprobar la validez de la

información del registro legal del programa, por ejemplo), debe huirse de un componente central de protección y crear partes completamente independientes que contengan todo el código necesario.

Antes de describir la comprobación de integridad en memoria, se describirá el tipo de algoritmo empleado en su cálculo.

Probablemente el algoritmo más extendido sea CRC, código de redundancia cíclica (en inglés, "Cyclic Redundancy Check"). Se aplica a un amplio abanico de aplicaciones actuales tanto en su formato CRC-32 (calcula un valor de 32 bits al realizar la comprobación) como en el de CRC-16 (calcula un valor de 16 bits).

Al emplear el algoritmo CRC-32, debe generarse primero la tabla de constantes que utilizará el algoritmo en sus cálculos. El código correspondiente se asemejará al siguiente:

```
void ...::Init_CRC32_Table()
{
   ULONG ulPolynomial = 0x04c11db7;// polinomio oficial
                                   // designado para
                                  // generar la tabla de
                                 // valores del
                                // algoritmo CRC-32
   for(int i = 0; i <= 0XFF; i++)
   { //generación de la tabla de valores
      CRC32_Table[i] = Reflect(i,8) << 24;
      for (int j = 0; j < 8; j++)
         CRC32_Table[i] = (CRC32_Table[i] << 1) ^
          ((CRC32_Table[i] &(1 << 31) ? ulPolynomial : 0);
      CRC32_Table[i] = Reflect(CRC32_Table[i],32);
   }
}
/******* se puede realizar el cálculo CRC-32 con la
función siguiente pero sin cumplir con el estándar
*******/
ULONG ...::Reflect(ULONG ref,char ch)
{
   ULONG value(0);
   for(int i = 1; i < (ch +1); i++)
   {
      if(ref & 1)
         value |= 1 << (ch - i));
      ref >>= 1;
   }
   return value;

    }
```

No obstante, no es estrictamente obligatorio emplear estas funciones: puede utilizarse una tabla de constantes ya generada. Dichas tablas se podrán incluir en la sección de referencias junto con el algoritmo CRC-32 en un programa ensamblador.

La función que realice las comprobaciones de integridad se asemejará a lo siguiente:

```
/*************************************************************/
/**Input: CheckedDataAddress                                 **/
/**      DataLength                                          **/
/**                                                         **/
/**                                                         **/
/*************************************************************/

/*******CRC-32*******/
DWORD Crc = 0xFFFFFFFF;
LPBYTE Buffer = (LPBYTE)CheckedDataAddress;

while(DataLength--)
{
   Crc =(Crc >> 8) ^ CRC32_Table[(Crc & 0xFF) ^ *Buffer++];
}
Crc = Crc ^ 0xFFFFFFFF;      // Crc = valor de 32 bits
                            //resultante de la comprobación
```

Este tipo de algoritmo de protección resulta innecesariamente complejo. Dependerá principalmente de la eficacia con la que se realice y oculte la comprobación de integridad. A un cracker no le interesará en absoluto lo maravilloso y complejo que sea un algoritmo de este tipo.

En la mayoría de los casos bastará con un algoritmo parecido al siguiente:

```
__asm
{
   mov ecx,data_length
   mov edi,start_from

   xor eax,eax                  // redefinición de los
                                // registros necesarios
   xor ebx,ebx
   xor esi,esi

Looop:
   mov al,byte ptr [edi]      // carga de datos
```

```
    mul esi                 // multiplicación
    add ebx,eax             // resultado a EBX
    inc edi                 // desplazándose al siguiente byte
    inc esi                 // aumenta ESI
    loop Looop              // bucle en la comprobación,
                            // contador = data_length
                            //--> EBX = comprobación
}
```

Puede servir este algoritmo de guía para crear uno propio.

En el ejemplo siguiente se muestra cómo realizar con sencillez la comprobación de integridad directamente en memoria. En él, se calcula la comprobación con áreas importantes del código de programa mediante el algoritmo anterior:

```
__asm
{
    mov eax,offset EndImpArea
    sub eax,offset ImportantArea          // EAX = longitud
                                          // de los datos
                                          // comprobados
    mov ecx,eax
    mov edi,offset ImportantArea          // dirección en
                                          // la que comenzará
                                          // la comprobación
    xor eax,eax                           // redefinición de los
                                          // registros necesarios
    xor ebx,ebx
    xor esi,esi

Looop:
    mov al,byte ptr [edi]                 // carga de datos
    mul esi                               // multiplicación
    add ebx,eax                           // resultados a EBX
    inc edi                               // desplazándose al
                                          // siguiente byte
    inc esi                               // aumenta ESI
    loop Looop
//--> EBX = comprobación
// no es preciso guardar el valor EBX - el código
siguiente no lo modifica //
```

```
/******* La única función del código siguiente consiste en
permitir generar la comprobación. Por tanto ni siquiera
resulta necesario ejecutarlo. Sustitúyanse estos datos con
los reales que el usuario quiera comprobar.*******/
ImportantArea:
   nop                                    // código
                                        // absolutamente inútil
   nop
   inc ebx
   dec ebx
   push eax
   pop eax
   nop
   nop

EndImpArea:
    cmp ebx,1463Fh      // comparación entre el valor
                        // correcto de la comprobación y el
                       //calculado
   je Ok
}
   MessageBox("El código del programa ha quedado
    modificado",NULL,MB_OK);
   return;

Ok:

    MessageBox("El código del programa no ha quedado
     modificado",NULL,MB_OK);
```

El algoritmo identificará todo intento de modificar el área de datos comprobada o de definir un punto de corte bpx. Es más, este algoritmo resulta bastante difícil de detectar por no utilizar ninguna función API. (La función `MessageBoxA` tan sólo se utiliza aquí como ejemplo, nunca debería emplearse en la protección.) Puesto que los crackers suelen utilizar puntos de corte bpm y bpr para encontrar estos tipos de protección, no debe olvidarse detectar su presencia.

En el CD adjunto se pueden encontrar algunos ejemplos donde se aplican comprobaciones de integridad con ficheros y en memoria empleando los algoritmos anteriormente mencionados.

Para quien no entienda cómo obtener el valor de comprobación correcto, bastará con aplicar a un depurador al algoritmo que efectúe el cálculo de la comprobación y observar el valor dado. Por supuesto, únicamente se procederá de esta manera cuando el área de datos objeto de la comprobación no se modifique con posterioridad.

Por último, ha de tenerse en cuenta qué hacer con el valor de comprobación calculado. No resultará suficiente realizar una sencilla comparación entre los valores correcto y el calculado puesto que resulta muy fácil de detectar y suprimir. Muchos mecanismos de seguridad modernos utilizan el valor calculado para descifrar los datos cifrados con el valor de comprobación correcto.

Otros métodos

Existen muchas otras formas de proteger un programa frente a la edición de su código. Las comprobaciones de integridad no constituyen el único método posible, sólo el más extendido (principalmente con las aplicaciones simples). Ya se ha hecho referencia a otros métodos alternativos que combinan distintas tecnologías de protección, por ejemplo, la modificación directa del código del programa mientras se está ejecutando.

CAPÍTULO 7

EL FORMATO PE Y SUS HERRAMIENTAS

DESCRIPCIÓN DEL FORMATO DE FICHERO PE

Probablemente resulte imposible determinar exactamente la procedencia de la especificación del formato de fichero PE, un formato de fichero nativo en todas las plataformas Win32. Seguramente provenga de un formato UNIX COFF (en inglés, "Common Object File Format", en castellano: "formato de fichero objeto común").

El formato PE (en inglés, "Portable Executable", en castellano, "ejecución portátil"), como su nombre indica, se aplica como formato universal a los ficheros ejecutables de todas las plataformas Win32. Las plataformas compatibles con este formato reconocerán y trabajarán con este tipo de fichero independientemente de la CPU en la que se ejecute el sistema operativo. Como todos los ficheros ejecutables de una plataforma Win32, a excepción de las librerías VxD y las DLLs de 16 bits, utilizan este formato, su estudio permitirá penetrar en los secretos de la programación del sistema.

El siguiente esquema muestra la estructura básica de un fichero PE:

La cabecera DOS MZ se incluye en el fichero sólo para mantener su compatibilidad con DOS. Cuando este fichero se ejecute en DOS, el sistema operativo lo considerará válido gracias a la cabecera DOS MZ y proseguirá su ejecución en la sección DOS. La inmensa mayoría de las actuales aplicaciones Win32 contienen aquí el famoso texto: "este programa no puede ejecutarse en modo DOS". Algunos programas antiguos conservan en esta sección su versión para DOS.

Cuando el sistema operativo reconoce el fichero con formato PE y lo procesa, el cargador PE busca en la cabecera DOS MZ la ubicación de la cabecera PE para ejecutarla saltándose la sección de DOS. Seguramente estará aquí almacenada la estructura más importante del fichero PE: la denominada `IMAGE_NT_HEADERS`. A continuación se describirá dicha estructura con mayor detalle.

El contenido real del fichero PE se divide en bloques de datos llamados secciones. Los datos no se dividen en bloques según sus atributos lógicos, sino sus atributos característicos, por ejemplo: sólo lectura, ejecutable, escribible, etc. Por lo tanto, no deben considerarse la secciones como bloques lógicos de datos, código, etc. Aunque en algunos casos ocurra así, téngase en cuenta que todo depende de los atributos. Puede resultar algo confuso, pero hay buenas razones para ello. Cuando se carga un fichero en memoria, se puede detectar con mucha rapidez aquellos atributos asignados a lugares concretos de la memoria.

Conforme el contenido del fichero se divida en secciones, será preciso guardar en algún lugar la información que las describa: su ubicación, tamaño, las características anteriormente mencionadas, etc. Éste constituye precisamente el objetivo de la tabla de secciones. Es un campo de estructuras, donde cada ítem de la estructura contiene información sobre una sección. De manera que si un fichero PE contiene, pongamos por ejemplo, cuatro secciones, este campo contendrá cuatro ítems (en circunstancias normales).

DESCRIPCIÓN Y FUNCIONAMIENTO DEL COMPRESOR–CODIFICADOR PE

Tras haber leído las secciones precedentes de este libro, el lector seguramente ya se habrá percatado de que el modo de protección más extendido contra el software pirata sea probablemente el uso de los compresores-codificadores PE. Se crearon a causa de la estructura compleja pero uniforme de los ficheros PE. Su diseño principal se basa en la codificación (o en la compresión) de secciones individuales de cualquier fichero PE y su descodificado en tiempo de ejecución, lo que imposibilita prácticamente la edición directa del código de programa. Al código de control, que, además de facilitar su descodificación, otorga integridad funcional al fichero prescindiendo de cualquier software complementario para su ejecución, se le pueden añadir elementos de protección complementarios no obligatorios. Algunos de los compresores-codificadores PE actuales se centran más en disminuir el tamaño que en proteger el programa.

Por tanto, se puede afirmar que los compresores-codificadores PE constituyen una protección compleja que permite añadir distintos elementos de protección a un programa existente. El fichero no podrá editarse mediante un editor hexadecimal (debido a la codificación o a la compresión), ni tampoco desensamblarlo (si bien, estrictamente hablando, sí se puede desensamblar, si bien el código codificado no tendrá ningún sentido) y resultará muy difícil depurar o volcar cuando se combine con otras técnicas de protección complementarias.

Como ya se indicó con anterioridad, el objetivo principal consiste en evitar la edición del programa. El resto normalmente constituye una protección contra el intento del cracker de descifrar manualmente el fichero para editar el programa. A los compresores-codificadores PE se les puede considerar protecciones de otras protecciones que ya existan dentro del programa

(como la comprobación de la presencia del CD, duración limitada, etc.). Existe un grupo especial de compresores-codificadores PE con sistemas de protección complejos (normalmente de tipo comercial) que ya incluyen estas protecciones. Al emplearlas, por ejemplo, se puede alterar un programa para que se convierta en una versión en demostración con duración limitada.

Al igual que sucedía con otro tipo de protecciones y programas, aún se puede llevar a cabo la edición, no sólo empleando un editor hexadecimal directamente, sino también sobrescribiendo la memoria. Existen muchos compresores-codificadores PE que resultan vulnerables por no considerar la posibilidad de que pueda modificarse el código de programa.

Creación de un codificador o compresor PE

En primer lugar, debe añadirse el código de control del compresor-codificador PE al fichero. No se debiera confiar, sin embargo, en el suficiente espacio libre que tenga una de las secciones para este propósito. La opción más sencilla (pero no la más segura) consiste en añadir una nueva sección del tamaño oportuno al fichero. No obstante, existen codificadores PE que buscan espacio libre para calcular la ubicación del fichero donde debiera añadirse el código, e incluso lo dividen en pequeñas secciones.

La situación cambia ligeramente con los diferentes tipos de compresores-codificadores. Algunos pretenden alcanzar la mejor compresión posible y reorganizar el fichero entero en sus mínimas partes; otros emplean un procedimiento completamente distinto.

Los datos tendrán que redirigirse al código de control una vez que quede incorporado al fichero. Frecuentemente es el propio código añadido el que se encarga de descifrar y realizar otras funciones para controlar el código original del programa. Si bien éste es el método más extendido, no representa en absoluto la única solución posible; por ejemplo, se pueden emplear varios hilos en el código de control.

La última cuestión que abordará esta sección atañe a las funciones importadas utilizadas por el programa. El fichero al que se añade el código de control no tiene necesariamente que importar todas las funciones que invoque: habrá que contemplar un método alternativo para importarlas.

Aquí concluye la descripción y funcionamiento general de un compresor-codificador PE típico.

Desventajas de los compresores-codificadores PE

Posiblemente la peor desventaja de los compresores-codificadores PE resida en su reversibilidad: tan sencillo como resulta su aplicación a un fichero, así de sencillo resulta su eliminación mediante un cierto tipo de descompresor-descodificador universal una vez el sistema de protección haya quedado anulado. Esto representa una desventaja frente a otros tipos de protección, puesto que resultará fácil y rápido suprimir la protección de un mayor número de programas codificados con el mismo compresor-codificador PE. Ahora bien, suprimir un tipo común de protección exige tanto tiempo como el empleado con un descodificador.

Todo depende principalmente de cómo se programe el compresor-codificador PE. Algunos de ellos emplean diversos métodos de generación aleatoria y procesamiento de, por ejemplo, el código de control, las rutinas descodificadoras, etc. Obstaculizando, incluso imposibilitando la programación de un descompresor-descodificador universal.

Algunos compresores-codificadores PE

ASPACK

Este programa de compresión alcanza, sin lugar a dudas, el mejor nivel de redacción de ficheros PE de entre todos los del mercado. Su algoritmo de compresión, cuyos resultados superan a los ficheros de tipo ZIP o RAR, constituye el núcleo de un compresor PE denominado ASProtect.

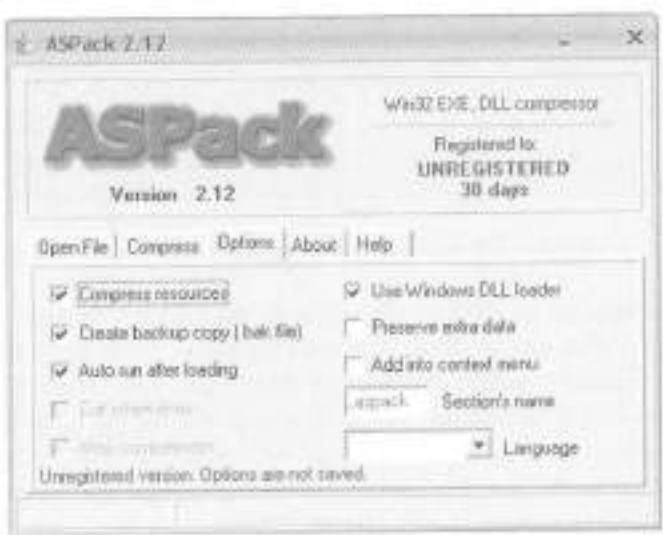

Figura 7-1. La interfaz de ASPack se asemeja mucho a la de ASProtect

El principal inconveniente de este compresor radica en la debilísima protección contra un depurador y la ausencia de defensas contra la edición del código memoria. Si además se considera que con un cargador se pueden superar fácilmente estos obstáculos, la protección del fichero comprimido se reducirá notablemente.

Al igual que ASProtect, ASPack demuestra una protección ligeramente superior al resto de la tabla de importaciones original, dificultando algo más los volcados de memoria. No obstante, un cargador evita realizar volcados de memoria.

CODESAFE

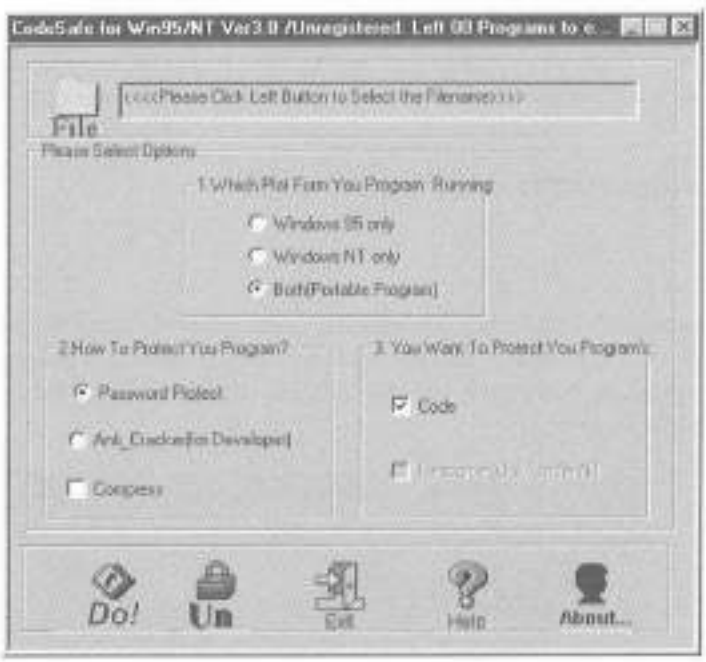

Figura 7-2. Las escasas alternativas del programa CodeSafe

Este compresor-codificador PE se puede utilizar con todas las plataformas Win32. Su comprensión dista mucho de las mejores y apenas se puede comparar con el compresor anterior. No obstante, incluye propiedades excelentes para dificultar la vulneración del fichero protegido. Las direcciones de las funciones importadas no se calculan mediante el método estándar de invocación de la función API, `GetProcAddress`, sino que se utiliza un método bastante nuevo e infrecuente a partir del formato PE. Su autor, Zhang De Hua, demostró ser un gran experto en este tipo de formato.

Aunque se comprueben puntos de corte bpx con los primeros cinco bytes de cada función importada en el fichero protegido, parece que el autor se olvidó de los tipos de puntos de corte restantes.

NEOLITE

Parecido a ASPack, el compresor NeoLite alcanza una compresión excelente. Desgraciadamente no se le puede pedir mucho más a este programa. La rutina de descompresión no contiene ningún elemento de protección, el propio programa puede inclusive descomprimir el programa comprimido. Todo ello demuestra que su objetivo primordial fue la reducción del tamaño del fichero PE más que su protección. Pudiera resultar útil para los desarrolladores profesionales de software que, con un concepto diferente de la protección, tan sólo exigen una compresión rápida y de calidad.

NFO

NFO es un codificador PE fabuloso, obviamente centrado en la protección de ficheros PE contra el cracking. Aplica un gran número de trucos antidepurador y antidesensamblador que bien pudiera costar algún dolor de cabeza al más experimentado cracker. Lástima que el fichero codificado sólo funcione con Windows 9x/Me, lo que restringe su uso significativamente.

PE-COMPACT

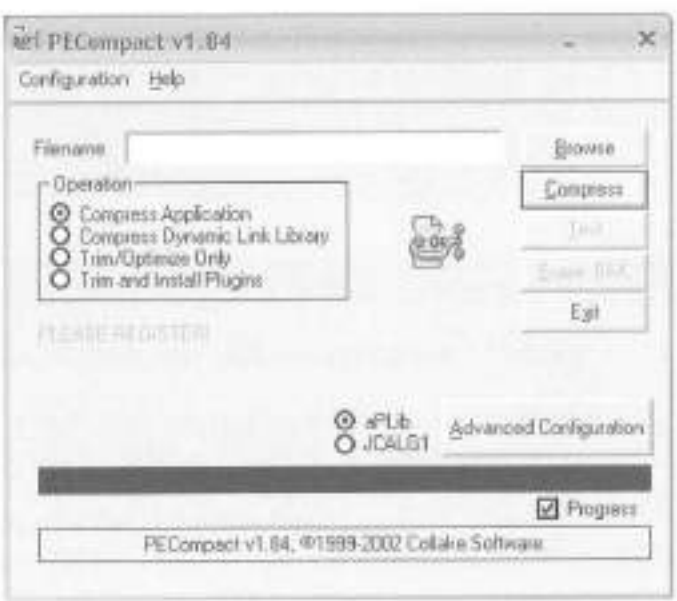

Figura 7-3. PE-compact

Éste es otro compresor PE más preocupado en la reducción del tamaño del fichero que en su protección. El programa utiliza dos algoritmos de compresión. El primero, basado en la librería Aplib, mundialmente conocida, y el segundo, creado por el autor de PE-compact, denominado JCALG. De libre distribución, obtiene muy buenos resultados y le coloca entre los mejores candidatos de los compresores del futuro.

La rutina de descompresión no incluye ningún elemento de protección, con lo que, como en los casos anteriores, resulta muy sencilla la descompresión manual.

PE-CRYPT

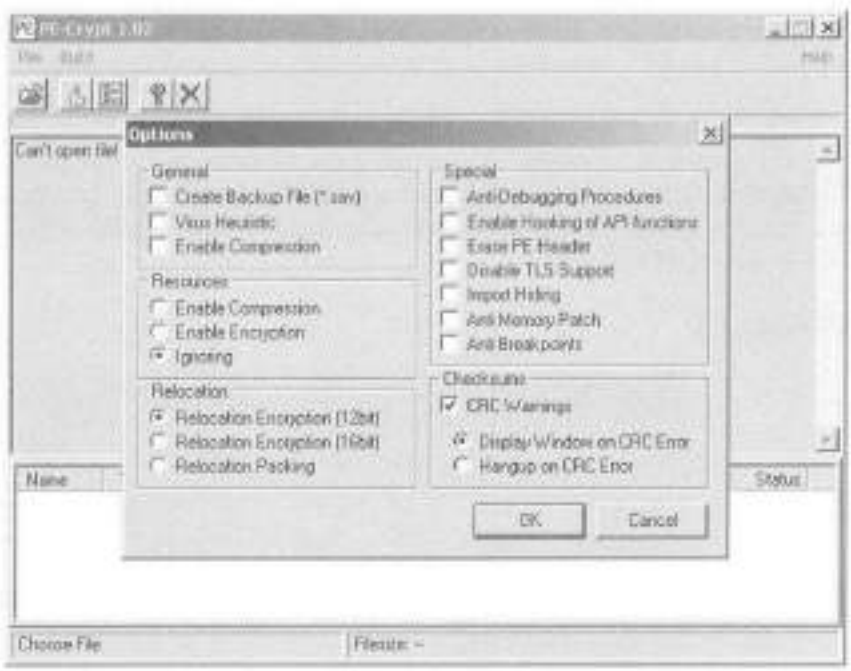

Figura 7-4. Las opciones de PE-Crypt contra el cracking

PE-Crypt ilustra bien el uso que se puede hacer del SMC. Este excelente compresor-codificador PE está enteramente basado en el SMC y lo aplica en gran medida. Su especial diseño imposibilita prácticamente cualquier intento de depuración o de desensamblaje, desesperando a quien lo intente. Es más, permite añadir a la rutina de descodificación otros algoritmos antidepuración o de supervisión de puntos de corte bpx y bpm.

El programa solventó los errores de sus versiones anteriores al añadir una opción que permitiera a ciertas rutinas evitar la edición del código del programa en memoria. Desgraciadamente, algunos de estos métodos no son compatibles con Windows

NT/2000/XP, con lo que los programas así protegidos sólo funcionarán con Windows 9x/Me.

Puesto que el código íntegro de la rutina de descodificación está programado con SMC, se complica muchísimo la tarea, anteriormente fácil, de detectar algoritmos antidepurador y antidesensamblador.

Desafortunadamente, incluso para este soberbio compresor-codificador PE existe descodificador. Se llama Bye PE Crypt. Ahora bien, si los autores, dos bien conocidos crackers, pusieran un poco más de esfuerzo en el desarrollo de este producto, PE Crypt volvería a ser indiscutiblemente uno de los mejores compresores-codificadores del mercado. Desgraciadamente, la ausencia de nuevas versiones sugiere que su desarrollo se ha paralizado.

PE-SHIELD

PE-Shield representa uno de los codificadores PE legendarios. Se hizo famoso al no existir durante mucho tiempo descodificador universal que lo anulase. Su excelente diseño estaba orientado precisamente a ello. Aplicaba métodos estándar de protección contra descodificadores genéricos, como ProcDump, algoritmos antirastreo (evitan rastrear la ejecución del programa) y, lo más importante, codificación polimórfica, empleado por vez primera en este codificador.

El bucle de descodificación también contiene rutinas de defensa frente a los puntos de corte (puntos de corte hardware incluidos) y contra los depuradores. El autor del programa, Anakin, un maestro en esta verdadera especialidad, creó dichas rutinas.

Para este codificador PE excepcional se creó un descodificador universal denominado UnPeShield. Ciertamente, su autor merece un gran respeto, puesto que programar un descodificador de análisis heurístico, en este caso no existe otra alternativa, no es tarea nada fácil.

PETITE

Figura 7-5. Cómoda interfaz de Petite

Debido a su alto precio, este compresor se utiliza principalmente en el sector comercial. Si bien no llega a la compresión alcanzada por el imbatible ASPack, obtiene muy buenos resultados.

Este producto cae más bien dentro de la categoría de herramientas de compresión más que de protección. Aunque la rutina de descodificación contenga algo parecido a un algoritmo antidepurador, su diseño no llega a los estándares actuales. No supone ningún obstáculo la descodificación manual del fichero protegido, lo que le descarta para realizar una buena protección.

SHRINKER

Menos por más. Así es cómo se podría describir este compresor-codificador PE de Blink Inc. No es que sea un mal producto. Consigue una compresión más que decente e incorpora un par de algoritmos antidepurador sencillos. Ahora bien, su precio no está a la altura de sus cualidades, resulta preferible optar por un compresor-codificador PE diferente. Cosa que no pone en duda ni siquiera su bien diseñada interfaz de usuario.

UPX

Figura 7-6. Opciones de UPX

Lento pero seguro. Los autores de este compresor-codificador PE se atuvieron a esta máxima y acertaron. Durante el increíble período de dos años, este producto se probó y modificó hasta que quedó lista su versión definitiva (no beta). Tantos esfuerzos obtuvieron su recompensa: UPX ("Empaquetador Definitivo para Ejecutables", en inglés "Ultimate Packer for Executables"), de merecido nombre, constituye el compresor-codificador PE más universal. No sólo soporta plataformas Win32, también DOS e incluso linux. Sus resultados al comprimir son equiparables a los de ASPack.

Lástima que sus autores invirtieran la mayor parte de su tiempo eliminando todo tipo de errores e inconvenientes de este programa (que, como contrapartida, le convierten en uno de los más fiables) y olvidaran en cierta medida la posibilidad de utilizarlo para algo más que comprimir. El que además de comprimir descomprima demuestra esta afirmación.

Este programa se ha hecho infame por su uso con troyanos y virus, reduciendo sus tamaños (característica muy útil cuando el virus se manda por correo electrónico) y dificultando mucho su detección.

WWPACK32

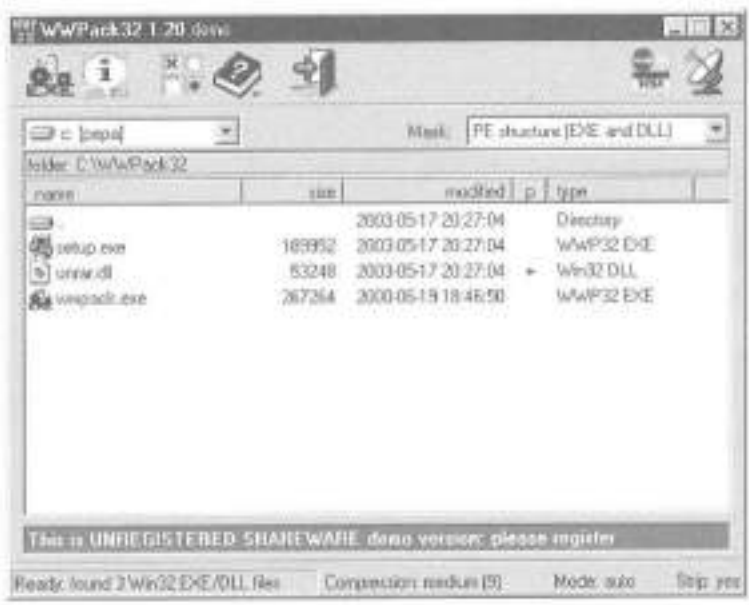

Figura 7-7. La interfaz de usuario de WWPack se asemeja a la de WinRAR

Este compresor PE no destaca entre los de calidad superior. Resulta evidente que los autores pusieron más empeño en el manejo del programa, simple, intuitivo y accesible incluso para un absoluto principiante. Desgraciadamente, carece de cualquier opción de seguridad. No es una gran cosa: el nivel de seguridad resulta prácticamente inexistente, y el único elemento de protección dado al fichero comprimido se reduce a unas simples rutinas de antidesensamblaje que pueden perfectamente controlarse con IDA.

La descompresión manual de fichero protegido es un juego de niños y tampoco es muy buena la compresión resultante.

FORMATO DE FICHERO PE

Comprobación del formato PE

Debido a las formidables dimensiones del formato PE, nunca se puede afirmar con rotundidad si un fichero cumple cabalmente el formato. Para ello, debería supervisarse la exactitud de todas las estructuras del fichero. Como es prácticamente imposible, sólo se suele comprobar el formato PE de las estructuras más importantes. Por supuesto, siempre dependerá del autor el grado de exactitud exigido con los resultados de la operación.

Probablemente el mejor compromiso en este caso consista en comprobar la parte más importante de un fichero PE, esto es, su cabecera PE. La cabecera contiene una estructura denominada `IMAGE_NT_HEADERS`, con el siguiente formato (extraído del fichero winnt.h):

```
typedef struct _IMAGE_NT_HEADERS {
 DWORD Signature;
 IMAGE_FILE_HEADER FileHeader;
 IMAGE_OPTIONAL_HEADER32 OptionalHeader;
}IMAGE_NT_HEADERS32,*PIMAGE_NT_HEADERS32;
```

- *Signature* contiene el valor 50h, 40h, 0, 0, es decir, la secuencia "PE" con ceros al final. Microsoft definió este valor como una constante denominada `IMAGE_NT_SIGNATURE`.

- La estructura *FileHeader* contiene información sobre la división física del fichero: nombre de secciones, información más detallada del fichero, etc.

- La estructura *OptionalHeader* contiene información sobre la división lógica del fichero PE, como la dirección de *Program Entry Point*, etc.

Si se deseara adquirir la estructura de `IMAGE_NT_HEADERS`, deberá invocarse la función `ImageNtHeader` definida en la librería imagehlp.dll dedicada a trabajar con ficheros PE. Lógicamente, primero habrá de cargarse el fichero pertinente en memoria. El proceso podría ser el siguiente:

```
PIMAGE_NT_HEADERS pImageNT;
DWORD NOBR;
HANDLE File =  CreateFile("nombre_del_fichero",GENERIC_READ,
 FILE_SHARE_READ,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,
 NULL);
                              //obtención del manejador del fichero
```

```
if (File == INVALID_HANDLE_VALUE)
{
     MessageBox("¡El fichero no puede abrirse para su
      lectura!", NULL,MB_OK |MB_ICONINFORMATION);
     return;
}

DWORD FileSize = GetFileSize(File,NULL);
BYTE *pMem = new BYTE [FileSize];    //  asignación de
                                     // memoria
ReadFile(File,pMem,FileSize,&NOBR,NULL);    //  carga del
                                    // fichero en memoria
pImageNT = ImageNtHeader((VOID*)pMem);  //obtención de la
                                     // estructura

IMAGE_NT_HEADERS
if (pImageNT == 0)    // ¿fichero PE?
{
     MessageBox("¡Fichero con formato PE incorrecto!",
      NULL,MB_OK |MB_ICONINFORMATION);
     CloseHandle(File);
     delete pMem;
     return;
}
MessageBox("¡Fichero con formato PE correcto!",NULL,MB_OK
 |MB_ICONINFORMATION);

CloseHandle(File);
delete pMem;
```

Si al invocar la función `ImageNtHeader`, se obtiene un resultado satisfactorio, el fichero cumplirá con el formato PE. En caso contrario, el fichero carecerá de la estructura y, por tanto, no resultará válido.

Ahora bien, de no conformarse con las innecesariamente complicadas, y con frecuencia poco eficaces, librerías de ayuda, utilícese el procedimiento siguiente. De nuevo resulta preciso obtener la estructura `IMAGE_NT_HEADERS` y comparar su variable `Signature` con la constante `IMAGE_NT_SIGNATURE`.

Ahora bien, ¿cómo encontrar la dirección de la estructura `IMAGE_NT_HEADERS`? La cabecera DOS MZ, definida por la estructura `IMAGE_DOS_HEADERS`, tiene el siguiente formato:

```
typedef struct _IMAGE_DOS_HEADER {    //  cabecera DOS .EXE
 WORD e_magic;    //  número mágico
```

```
WORD e_cblp;    //  bytes de la última página del fichero
WORD e_cp;    //  páginas del fichero
WORD e_crlc;    //  reubicaciones
WORD e_cparhdr;    //  tamaño de la cabecera en los
                                                    // párrafos
WORD e_minalloc;    //  mínimos párrafos extra necesarios
WORD e_maxalloc;    //  máximos párrafos extra necesarios
WORD e_ss;    //  valor SS inicial (relativo)
WORD e_sp;    //  valor SP inicial
WORD e_csum;    //  comprobación
WORD e_ip;    //  valor IP inicial
WORD e_cs;    //  valor CS inicial (relativo)
WORD e_lfarlc;    //  dirección del fichero de la tabla de
                                                // reubicaciones
WORD e_ovno;    //  número de superposición
WORD e_res [4];    //  palabras reservadas
WORD e_oemid;    //  identificador OEM (para e_oeminfo)
WORD e_oeminfo;    //  información OEM; propio de e_oemid
WORD e_res2 [10];    //  palabras reservadas
LONG e_lfanew;    //  dirección del fichero de la nueva
                                                // cabecera exe
}IMAGE_DOS_HEADER,*PIMAGE_DOS_HEADER;
```

La última variable `e_lfanew` almacena la dirección de la cabecera PE del fichero. La primera palabra de la estructura `IMAGE_NT_HEADERS` guarda los valores 4Dh, 5Ah, esto es, "MZ" (esto es, la cabecera DOS MZ), también definida por Microsoft como la constante para `IMAGE_DOS_SIGNATURE`, es decir, en la variable `e_magic`, y por lo tanto también en la primera palabra de todo el fichero.

Si se desea una seguridad absoluta, se podrá probar la validez de la estructura `IMAGE_DOS_HEADER` comparando la primera palabra del fichero con la constante anteriormente mencionada y entonces utilizar la dirección de la cabecera PE a partir de su variable `e_lfanew`.

```
DWORD NOBR;
LPBYTE Buffer;

HANDLE File =
 CreateFile("nombre_del_fichero",GENERIC_READ,FILE_SHARE
 _READ,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
                    //obtención del manejador del fichero
 if (File ==INVALID_HANDLE_VALUE)
{
 MessageBox("¡El fichero no se puede abrir en modo
  lectura!",NULL,MB_OK | MB_ICONINFORMATION);
```

```
  return;
}

DWORD FileSize = GetFileSize(File,NULL);
BYTE *pMem = new BYTE [FileSize];    // asignación de
                                     // memoria
ReadFile(File,pMem,FileSize,&NOBR,NULL);    // carga
                                     // del fichero en memoria
Buffer = pMem;

if (*((LPWORD)Buffer) == IMAGE_DOS_SIGNATURE)   //  ¿MZ?
{
 LPDWORD Signature =
  (LPDWORD)(Buffer+*((LPDWORD)Buffer+15));

 if (*Signature != IMAGE_NT_SIGNATURE)    //  ¿PE?
       MessageBox("¡Fichero con formato PE
        incorrecto!",NULL,MB_OK |MB_ICONINFORMATION);
 else
       MessageBox("¡Fichero con formato PE correcto!",
        NULL,MB_OK |MB_ICONINFORMATION);
 }
 else
       MessageBox("¡Fichero con formato PE
        incorrecto!",NULL,MB_OK |MB_ICONINFORMATION);

CloseHandle(File);
delete pMem;
```

Cabecera PE

Hasta ahora se ha descrito brevemente la arquitectura de la cabecera PE en las secciones anteriores de este capítulo. Se crea mediante la estructura `IMAGE_NT_HEADERS`, donde juega un papel señalado la variable `Signature`. Seguidamente se examinará el resto de esta estructura. Consta de dos estructuras principales denominadas `FileHeader` y `OptionalHeader`. A continuación se presenta la arquitectura de la primera:

```
typedef struct _IMAGE_FILE_HEADER {
 WORD Machine;
 WORD NumberOfSections;
 DWORD TimeDateStamp;
 DWORD PointerToSymbolTable;
 DWORD NumberOfSymbols;
 WORD SizeOfOptionalHeader;
```

```
 WORD Characteristics;
}IMAGE_FILE_HEADER,*PIMAGE_FILE_HEADER;
```

`Machine` indica el nombre de la máquina. En el caso de la plataforma Intel este valor equivale a la constante `IMAGE_FILE_MACHINE_I386`, esto es, 14Ch.

`NumberOfSections` contiene el número de secciones del fichero. Recuérdese bien esta variable cuando, en la parte del capítulo dedicada a los codificadores PE, deba añadirse una sección nueva al fichero.

Las siguientes tres variables carecen aquí de interés. En `TimeDateStamp` se guarda la fecha y la hora de creación del fichero, `PointerToSymbolTable` y `NumberOfSymbols` se utilizan en la depuración.

`SizeOfOptionalHeader`, como su nombre sugiere, indica el tamaño de la estructura `OptionalHeader`.

`Characteristics`: características del fichero, por ejemplo, si es un fichero EXE o DLL.

La última parte de la estructura `IMAGE_NT_HEADERS` consiste en una estructura denominada `OptionalHeader`. Por ser muy larga, sólo se destacarán sus partes más importantes:

```
typedef struct _IMAGE_OPTIONAL_HEADER {
 //
 //  Campos estándar.
 //
...
 DWORD AddressOfEntryPoint;
...
 //
 //  Campos NT complementarios.
 //
 DWORD ImageBase;
 DWORD SectionAlignment;
 DWORD FileAlignment;
 WORD MajorOperatingSystemVersion;
 WORD MinorOperatingSystemVersion;
 ...
 DWORD SizeOfImage;
 DWORD SizeOfHeaders;
 DWORD CheckSum;
 WORD Subsystem;
```

```
...
IMAGE_DATA_DIRECTORY DataDirectory
  [IMAGE_NUMBEROF_DIRECTORY_ENTRIES];

}IMAGE_OPTIONAL_HEADER32,*PIMAGE_OPTIONAL_HEADER32;
```

- `AddressOfEntryPoint` (posteriormente denominado `Program Entry Point`) es una RVA (véase más adelante) de la primer instrucción del código de programa que arrancará su ejecución.

- `ImageBase` representa la dirección de carga preferida para el fichero PE. En la mayoría de las ocasiones este valor es 400000h.

La dirección de carga preferida constituye la dirección de arranque en la que el cargador PE intentará cargar el fichero en memoria. De manera que si el valor fuera 400000h, el fichero quedará cargado en un espacio de memoria a partir de la dirección 00400000. El cargador PE sólo será capaz de cargar el fichero en esta dirección siempre que un proceso distinto no esté ya empleando este espacio de direcciones. Si fracasara, se elegiría una dirección nueva para cargar el fichero memoria. El fichero permanecerá operativo incluso tras esta operación gracias a la RVA, nueva sigla cuyo significado se describirá posteriormente.

Con frecuencia se piensa que vivimos en un mundo de sistemas operativos basados en la multitarea. Debe precisarse que aunque el usuario pueda pensar que se están ejecutando más operaciones a la vez, la realidad es bien distinta. El sistema tan sólo salta de una a otra muy rápido (lo que constituye una explicación simplificada. Si se deseara más información sobre este asunto, Internet está repleto de material al respecto). Esto explica que el espacio de direcciones referido por `ImageBase` no esté ya ocupado al arrancar el primer programa y le parezca al usuario que los procesos individuales compartan sus recursos (por ejemplo, que cinco recursos utilicen la misma memoria).

- `SectionAlignment` determina el alineamiento de las secciones en memoria. Normalmente se define con el valor de 1000h, lo que indica que cada sección debe comenzar en una dirección de memoria múltiplo de 1000h. Por lo tanto, si la primera sección comenzara en la dirección 00407000 por ejemplo, aunque su tamaño fuera de un solo byte, la segunda sección empezaría en la dirección 00408000.

- `FileAlignment` se parece a `SectionAlignment`, la única diferencia reside en que determina el alineamiento físico de las secciones individuales directamente en el fichero en vez de en memoria. Normalmente tiene el valor de 200h.

- `MajorSubsystemVersion` y `MinorSubsystemVersion`, de más importancia aquí, determinan la versión del subsistema Win32.

- `SizeOfImage` contiene el tamaño total de la imagen del fichero tras haberse cargado en memoria, se define como la suma de todas las cabeceras y secciones, alineamiento incluido.

- `SizeOfHeaders` representa la suma de todas las cabeceras (incluyendo la sección DOS) y tabla de secciones. Constituye, por tanto, la ubicación de la primera sección en el fichero.

- `Subsystem` indica el subsistema NT al que va destinado el fichero. La mayoría de los programas Win32 definen el valor Windows GUI o Windows CUI (aplicación de consola, en inglés "console application").

- `DataDirectory`, campo de la estructura `IMAGE_DATA_DIRECTORY` con RVAs de estructuras importantes del fichero PE, por ejemplo, las tablas de importación o exportación.

Tabla de secciones

La tabla de secciones está representada en el fichero PE por estructuras `IMAGE_SECTION_HEADER`. El número de ítems en este campo y, por tanto, también el número de estas estructuras se guarda en la variable `NumberOfSections` de la estructura `IMAGE_FILE_HEADER`.

A continuación se describe la arquitectura de la estructura `IMAGE_SECTION_HEADER`:

```
typedef struct _IMAGE_SECTION_HEADER {
 BYTE Name [IMAGE_SIZEOF_SHORT_NAME];
 union {
       DWORD PhysicalAddress;
       DWORD VirtualSize;
 }Misc;
 DWORD VirtualAddress;
 DWORD SizeOfRawData;
 DWORD PointerToRawData;
 DWORD PointerToRelocations;
 DWORD PointerToLinenumbers;
 WORD NumberOfRelocations;
 WORD NumberOfLinenumbers;
 DWORD Characteristics;
}IMAGE_SECTION_HEADER,*PIMAGE_SECTION_HEADER;
```

- `Name` contiene un máximo de ocho caracteres, indica el nombre de la sección y no puede terminar en un carácter nulo.
- `Union Misc` está, de hecho, representado por la única palabra doble, `VirtualSize`. Esta variable contiene el tamaño de la sección en memoria sin alineamiento, según el valor `SectionAlignment`.
- `VirtualAddress` determina el valor RVA de la sección.
- `SizeOfRawData` determina el tamaño real de la sección en el fichero, incluyendo el alineamiento `FileAlignment`.
- `PointerToRawData` es un puntero al principio de la sección correspondiente en el fichero.
- `Characteristics` describe las características de los datos de la sección, por ejemplo, de sólo lectura, lectura y escritura, código ejecutable, etc.

DIRECCIONES VIRTUALES, MATERIALES Y VIRTUALES RELATIVAS (RVA)

La palabra virtual se refiere a información que indica la posición en memoria. Material, por otro lado, indica la ubicación directamente en el fichero.

Se ilustrará todo con el siguiente ejemplo. Ábrase con un editor PE (uno de los mejores se incluye en el CD adjunto) el fichero notepad.exe (se asume que todo el mundo tiene este fichero en su disco). Se obtendrán los valores siguientes:

Name	VirtualSize	VirtualAddress (Desplazamiento Virtual)	SizeOfRawData (Tamaño material)	PointerToRawData(Desplazamiento material)
.text	000065CAh	00001000h	00006600h	00000600h
.data	00001944h	00008000h	00000600h	00006C00h
.rsrc	00006000h	0000A000h	00005400h	00007200h

Ábrase ahora el fichero con algún editor hexadecimal para situarse en la ubicación (desplazamiento) `PointerToRawData`, esto es, 600h. Esta nueva situación señalará el principio de la primera sección del fichero. Si se añade a este valor el tamaño de la propia sección (`SizeOfRawData`), es decir, 600h + 6600h, la ubicación lógica será el principio de la segunda sección denominada .data (6C00h). Muy sencillo.

Ahora bien, no resulta tan simple en el caso de la memoria. `ImageBase`, la dirección de carga preferida, causa ciertas ambigüedades que confunden a muchos principiantes.

Seguramente se recordará que este valor sólo se utilizará si no existiera ya un proceso distinto utilizando ese espacio específico de direcciones. Si no se consiguiera la asignación de memoria indicada por esta dirección, deberá emplearse una dirección diferente. Imagínese por un momento que todos los valores importantes del formato PE apuntan directamente a una dirección específica de la memoria (como sucede con las variables materiales de este fichero). El cargador PE tendría entonces que sustituirlas todas por otras nuevas direcciones.

Es aquí donde se utiliza la dirección virtual relativa (RVA, en inglés "Relative Virtual Address"), sirve de puntero a la ubicación en memoria sin emplear el valor de `ImageBase`. Tras sumar el valor de RVA al de `ImageBase`, se obtiene una ubicación real de memoria. La RVA se convierte entonces en VA (dirección virtual, en inglés "Virtual Address", no se confunda con la variable `VirtualAddress`, que de hecho es una RVA), con el valor real de la dirección en memoria (lo que resulta bastante lógico al omitir la palabra "relativa"). Todas las variables importantes del formato PE que describen posiciones de memoria (por ejemplo, `VirtualAddress`) contienen de hecho RVAs de estas posiciones de memoria.

No obstante, aún no se han resuelto todos los problemas. Muchas instrucciones del fichero se refieren a valores absolutos de direcciones en memoria; por lo tanto, una nueva dirección de carga del fichero en memoria inutilizaría el fichero. Ésta es la razón de que el formato PE defina las llamadas reubicaciones; así, al corregir todos estos valores según la nueva dirección de carga, el fichero podrá funcionar con normalidad.

Las reubicaciones se utilizan primordialmente con las librerías DLL, apenas tienen uso práctico con ficheros EXE. En circunstancias normales, debido a la multitarea, sólo se ejecuta un proceso, por lo que resulta prácticamente imposible ocupar la memoria señalada por el valor de `ImageBase` del fichero EXE creador del proceso. Por otra parte, muchas librerías DLL pueden ejecutarse dentro de un proceso (funciones de importación): así sí resulta posible ocupar `ImageBase`. Circunstancia solucionada mediante las reubicaciones. La única excepción en el caso de ficheros EXE se reduce a los casos en que exporten funciones. Si las funciones se importasen desde el fichero por el proceso, la situación resultaría muy similar a la de las librerías DLL; caso también resuelto con las reubicaciones.

Si la `ImageBase` de un fichero EXE del proceso específico resultase ocupada, revelará normalmente a algún otro error, que posiblemente colapse el programa incluso el sistema operativo.

Algunos lenguajes de programación, como Delphi, añaden por omisión reubicaciones dentro de cada uno de sus programas: casi siempre se encuentran reubicaciones en los programas creados con estos lenguajes.

Volvamos al ejemplo del Bloc de notas (notepad.exe). Si se desease examinar el principio de la primera sección de este programa en memoria con un depurador, añádase el valor de `ImageBase` (en este caso 1000000h) a la dirección virtual (en este caso 65CAh), se obtendrá el valor 10065CAh, es decir, la dirección real de la primera sección del programa en memoria.

RVA, suma de la primera sección (`VirtualAddress`) con su tamaño (`VirtualSize`), contiene el valor 75CAh, que constituye la RVA de la posición en memoria donde termina la primera sección. Para obtener el principio de la siguiente sección de memoria, basta con alinear la sección según el valor `SectionAlignment` (en este caso, 1000h), cuyo resultado es 8000h.

En ocasiones, diversas secciones del fichero no se alinean según `ImageBase`. Los cálculos se vuelven entonces mucho más complicados y confusos. Está de más explicar aquí cómo llevar a cabo estos cálculos. Resulta mucho más fácil utilizar algún programa que los ejecute. Dos de ellos se encuentran en el CD incluido en este libro.

Tabla de importaciones

La tabla de funciones importadas o, más brevemente, importaciones, y especialmente las funciones de importación en sí, constituye una de las piedras angulares de la arquitectura de la plataforma Win32. La sola palabra "importar" indica claramente el cometido de estas funciones.

Una función importada consiste en una función invocada por el fichero PE, sin que el propio fichero la contenga. La tabla de importaciones del fichero contendrá toda la información necesaria para emplear las funciones importadas (nombre de la función, librería DLL, etc.) pero no la propia función.

Para que un fichero PE importe una función, otro fichero PE debe exportarla. Normalmente las funciones se suelen exportar mediante librerías DLL, ciertamente muy extendidas.

El último campo de la estructura `IMAGE_OPTIONAL_HEADER`, contenida dentro de `IMAGE_NT_HEADERS` incluye un campo de dieciséis estructuras `IMAGE_DATA_DIRECTORY` denominado `DataDirectory`. Cada una de estas

estructuras contiene información sobre el tamaño y RVA de algunas posiciones importantes del fichero. A continuación se expone la arquitectura de la estructura IMAGE_DATA_DIRECTORY:

```
typedef struct _IMAGE_DATA_DIRECTORY {
 DWORD VirtualAddress;
 DWORD Size;
}IMAGE_DATA_DIRECTORY,*PIMAGE_DATA_DIRECTORY;
```

VirtualAddress es la RVA de la estructura correspondiente, y Size, su tamaño.

A partir de los datos contenidos en la tabla siguiente se pueden determinar fácilmente los ítems más importantes del campo DataDirectory de estructuras IMAGE_DATA_DIRECTORY específicas y la información sobre los datos que contengan:

Ítem del campo	Información
0	Tabla de exportaciones
1	Tabla de importaciones
2	Recursos
3	Excepción
5	Reubicación base
6	Depuración
7	Derechos de autor
9	Tabla TLS
10	Cargar configuración
11	Vínculo de importación

La segunda estructura de `DataDirectory` contiene información sobre la tabla importaciones. El valor de `VirtualAddress` en esta estructura será la RVA de la tabla de importaciones. Así se resuelve la búsqueda de la tabla de importaciones en el fichero. A continuación se examinará su estructura.

La tabla importaciones consta de estructuras `IMAGE_IMPORT_DESCRIPTOR`. A continuación se presenta la arquitectura de esta estructura:

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
 union {
       DWORD Characteristics;
       DWORD OriginalFirstThunk;
 };
 DWORD TimeDateStamp;
 DWORD ForwarderChain;
 DWORD Name;
 DWORD FirstThunk;
}IMAGE_IMPORT_DESCRIPTOR;
```

Finaliza con una estructura llena de caracteres nulos.

Cada estructura `IMAGE_IMPORT_DESCRIPTOR` contiene información sobre la librería desde la que se importarán las funciones. De manera que si el fichero importa, pongamos por caso, diez librerías, la tabla de importaciones `IMAGE_IMPORT_DESCRIPTOR` contendrá diez estructuras más una estructura final rellena con caracteres nulos (siempre que no se enlacen mal).

El primer ítem de la estructura `IMAGE_IMPORT_DESCRIPTOR` es `Union`, que será sustituido por una variable de doble palabra: `OriginalFirstThunk`. Esta variable contiene la RVA de las estructuras `IMAGE_THUNK_DATA`. El campo finaliza, al igual que con `IMAGE_IMPORT_DESCRIPTOR`, con una estructura rellena de caracteres nulos.

`Name` es la RVA del nombre de la librería, esto es, el puntero RVA del nombre de la librería en formato ASCII.

`FirstThunk` incluye la RVA de la segunda estructura `IMAGE_THUNK_DATA`, idéntica a la señalada por `OriginalFirstThunk`. La diferencia entre estos dos campos reside en la sustitución realizada a los ítems en el campo apuntado por `FirstThunk` con las direcciones reales de las funciones importadas cuando el cargador PE procesa las funciones importadas. El segundo campo donde apunta `OriginalFirstThunk` permanece invariable cuando el cargador PE necesita encontrar los valores originales de las estructuras `IMAGE_THUNK_DATA`.

La estructura IMAGE_THUNK_DATA ni siquiera debiera llamarse estructura puesto que sólo contiene el puntero RVA a otro estructura, IMAGE_IMPORT_DESCRIPTOR. Para evitar la confusión, resulta más preciso concebir esta estructura como un puntero a la estructura IMAGE_IMPORT_BY_NAME. El número de ítems en las estructuras IMAGE_THUNK_DATA es lógicamente idéntico al número de ítems de las estructuras IMAGE_IMPORT_BY_NAME, que determinan el número de funciones importadas desde una librería.

Finalmente en la estructura IMAGE_IMPORT_BY_NAME reside la información sobre la función importada. A continuación se expone la arquitectura de esta estructura:

```
typedef struct _IMAGE_IMPORT_BY_NAME {
 WORD Hint;
 BYTE Name [1];
}IMAGE_IMPORT_BY_NAME,*PIMAGE_IMPORT_BY_NAME;
```

Hint no tiene aquí uso práctico. Señala la tabla de exportaciones de la librería desde la que se importa la función; el cargador PE lo utiliza para acelerar la búsqueda de las funciones importadas en la tabla de exportaciones de la librería.

Name contiene el nombre de la función en formato ASCII. Con tan pequeño espacio de almacenamiento no bastaría para guardar el nombre de la función, razón por la que Name tiene en realidad tamaño flexible.

La situación se complica más cuando algunas funciones importadas carecen de estructura IMAGE_IMPORT_BY_NAME. A estas funciones se las denomina funciones ordinales: no se las importa según su nombre, sino su posición. En este caso, la estructura IMAGE_THUNK_DATA no señala a la estructura IMAGE_IMPORT_BY_NAME. A estas funciones se las denomina funciones ordinales: no se las importa según su nombre, sino que su palabra final señala la posición de la función cambiando a 1 su bit más significativo ("MSB", en inglés).

Para probar con facilidad el MSB de doble palabra se definió la constante IMAGE_ORDINAL_FLAG32 con el valor 80000000h. Por lo tanto, si la posición de la función ordinal fuera ABCDh, IMAGE_THUNK_DATA tendrá el valor 8000ABCDh.

Tabla de exportaciones

La parte anterior de este capítulo se centró en las funciones importadas. Sus contrarias son las exportadas. Su principio resulta muy sencillo. Si se desea importar funciones con un fichero PE, otro fichero PE tendrá que exportarlas.

Al igual que con las importadas, las funciones se pueden exportar según su nombre o ubicación (funciones ordinales).

Considerando que el desarrollador apenas puede influir directamente en la ubicación de estas funciones, como en una librería DLL ya creada, la exportación o importación de las funciones ordinales puede resultar un poco laboriosa.

La tabla de exportaciones puede localizarse, al igual que la tabla de importaciones, mediante el campo `DataDirectory` de la estructura `IMAGE_DATA_DIRECTORY`. La tabla queda definida por la estructura `IMAGE_DATA_DIRECTORY`:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
 DWORD Characteristics;
 DWORD TimeDateStamp;
 WORD MajorVersion;
 WORD MinorVersion;
 DWORD Name;
 DWORD Base;
 DWORD NumberOfFunctions;
 DWORD NumberOfNames;
 DWORD AddressOfFunctions;
 DWORD AddressOfNames;
 DWORD AddressOfNameOrdinals;
}IMAGE_EXPORT_DIRECTORY,*PIMAGE_EXPORT_DIRECTORY;
```

`AddressOfFunctions` contiene la RVA del campo de direcciones de las funciones exportadas individuales. El número de items en este campo equivale a `NumberOfFunctions`.

`AddressOfNames` contiene la RVA del campo de RVAs de los nombres de las funciones exportadas. El número de items en este campo equivale a `NumberOfNames`.

Si el fichero no exportase ninguna función ordinal, `NumberOfFunctions` y `NumberOfNames` tendrían el mismo valor. Caso contrario, la diferencia entre `NumberOfFunctions` y `NumberOfNames` sería igual al número de funciones ordinales.

`AddressOfNameOrdinals` señala a un campo de índices que interconecta los campos de nombres y de direcciones. Al representar este campo de índices cierto tipo de conexión entre los nombres de las funciones y sus direcciones, no resulta aplicable a las funciones ordinales.

El campo al que señala `AddressOfNameOrdinals` contiene un número de items equivalente al campo de nombres. Cada item de este campo se relaciona con el

mismo ítem del campo de nombres. Razón por la que ambos campos deben procesarse simultáneamente. Una dirección se empareja a siempre con un nombre de función, pero no necesariamente ocurre lo contrario.

El siguiente algoritmo puede utilizarse para calcular la RVA de una función exportada:

```
//  EBX = AddressOfNameOrdinals
mov dx,[ebx]    //  DX contiene el índice de las
                // posiciones en el campo de direcciones
movzx edx,dx    //  completado con ceros
shl edx,2    //  *4
add edx,AddressOfFunctions    //  EDX = RVA
```

Si se intentara colocar este algoritmo en un bucle con objeto de calcular las RVAs de todas las funciones exportadas, sería preciso aumentar el valor del puntero `AddressOfNameOrdinals` al campo de índices según el tamaño y del ítem de este campo, esto es, una palabra. Por supuesto, `AddressOfNameFunctions` no se ve aumentado.

CONFIGURACIÓN DE UN CODIFICADOR PE

Esta parte del capítulo estará dedicada a describir la programación de un codificador PE para que el lector pueda crear el suyo propio (Win32 EXE). Si se requiriese emplear el codificador con otros ficheros PE, por ejemplo DLLs, sería preciso alterar el procedimiento de creación del codificador en muchos aspectos. En este caso pudiera ser útil el algoritmo de reubicación incluido en la sección de referencia de este libro.

Los ejemplos individuales se mostrarán de forma paulatina para poder crear el codificador PE paso a paso.

A la hora de crear un codificador PE, siempre será necesario decidir dónde y cómo se añadirá el código de control del codificador. No ha de confiarse en que una sección tenga suficiente espacio para albergar este código. Resultará preferible una solución más sencilla: añadir al fichero una sección nueva del tamaño oportuno.

Inclusión de una sección nueva en un fichero

En primer lugar, debe cargarse el fichero en memoria y obtener el puntero a la estructura `IMAGE_NT_HEADERS`.

```
PIMAGE_NT_HEADERS pImageNT;
DWORD NOBR;

HANDLE File =
 CreateFile("nombre_del_fichero",GENERIC_READ,FILE_SHARE
  _READ,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
                  // obtención del manejador del fichero
                  // El fichero se abre con parámetro de
                  //lectura, las modificaciones se
                 //guardarán en otro fichero

if (File == INVALID_HANDLE_VALUE)
{
 MessageBox("¡El fichero no se puede abrir en modo
  lectura!",NULL,MB_OK |MB_ICONINFORMATION);
 return;
}

DWORD FileSize = GetFileSize(File,NULL);
BYTE *pMem = new BYTE [FileSize];   //  asignación de
                                    // memoria
for (DWORD i = 0; i < FileSize; i++)      //  por si se
                                  // "borrase" la memoria
{
 pMem [i] = 0;
}

ReadFile(File,pMem,FileSize,NOBR,NULL);   //  carga del
                                   // fichero en memoria
pImageNT = ImageNtHeader((VOID*)pMem);    //  obtención
                                   //de la estructura
                                   //IMAGE_NT_HEADERS
if (pImageNT == 0)
{
 MessageBox("¡Fichero con formato PE
  incorrecto!",NULL,MB_OK |MB_ICONINFORMATION);
 CloseHandle(File);
 delete pMem;
 return;
}
```

A partir de este momento, se puede añadir una función nueva denominada `AddSection` para añadir una nueva sección. Como primer paso, esta función tiene que ver si hay suficiente espacio en la tabla de secciones para incluir una más. Comprobación muy sencilla. Basta con calcular los tamaños de todas las cabeceras, las definiciones de las secciones individuales de la tabla de secciones más el espacio necesario para añadir la

definición de la nueva sección. Compárese entonces el resultado con el valor de SizeOfHeaders. Si fuera inferior que la suma resultante, obviamente no quedaría suficiente espacio libre en la tabla de secciones para definir una nueva. La diferencia entre estos dos valores representa el tamaño del espacio libre en la tabla de secciones.

```
DWORD CPEcoderDlg::AddSection(BYTE *pMem,
PIMAGE_NT_HEADERS pImageNT)
{
 DWORD SecNum = pImageNT->FileHeader.NumberOfSections;
                                    // número de secciones
 DWORD SecSize = sizeof IMAGE_SECTION_HEADER;
              // tamaño de la definición de una sección

 DWORD Size = (SecNum+1)*SecSize;    //  tamaño nuevo de
                 // definición en la tabla de secciones
 DWORD TotalSize = (DWORD)pImageNT-
  (DWORD)pMem+Size+sizeof IMAGE_NT_HEADERS;    //  suma
                                  // de los tamaños de todas
                            // las cabeceras y definiciones
                                       // de las secciones
 DWORD HeadersSize = pImageNT-
  >OptionalHeader.SizeOfHeaders;
 if (TotalSize >HeadersSize)     // ¿espacio suficiente
                                 //para una sección nueva?
 {
      MessageBox("¡espacio insuficiente en la tabla de
       secciones!",NULL,MB_OK |MB_ICONINFORMATION);
      return 0;
 }
```

Al saber que se deberá escribir posteriormente en la mayor parte de las secciones (para codificar y descodificar datos principalmente), será necesario alterar sus características para permitir su escritura (mediante la función lógica OR 80000000h).

También será preciso localizar cierta información sobre la última sección del fichero para poder colocar detrás la nueva. Todo ello se realizará de golpe mediante la siguiente función:

```
IMAGE_SECTION_HEADER
 CPEcoderDlg::RvaToSection(PIMAGE_NT_HEADERS
 pImageNT,BYTE*pMem,DWORD AdrOfSecTable,BOOL AdjustChar)
{
 PIMAGE_SECTION_HEADER pSection;
 IMAGE_SECTION_HEADER Section;
```

```
DWORD vaddr;    //  VirtualAddress
memcpy(&vaddr,(VOID*)(AdrOfSecTable+12),sizeof DWORD);
pSection = ImageRvaToSection(pImageNT,pMem,vaddr);
if (pSection == 0)
{   // no se procesó esta sección con la función
   // ImageRvaToSection
      DWORD vsize,raddr,rsize,characteristics;
      memcpy(&vsize,(VOID*)(AdrOfSecTable+8),sizeof
       DWORD);
      memcpy(&rsize,(VOID*)(AdrOfSecTable+16),sizeof
       DWORD);
      memcpy(&raddr,(VOID*)(AdrOfSecTable+20),sizeof
       DWORD);
      memcpy(&characteristics,(VOID*)(AdrOfSecTable+36)
       ,sizeof DWORD);
      Section.VirtualAddress = vaddr;
                                          //VirtualAddress
      Section.Misc.VirtualSize = vsize; //VirtualSize
      Section.PointerToRawData = raddr;
                                        //PointerToRawData
      Section.SizeOfRawData = rsize;    //SizeofRawData
      if (AdjustChar)    //  modificación de las
                         // características para permitir
                                          //escritura
          Section.Characteristics = characteristics |
           0x80000000;

      return Section;
}
      if (AdjustChar)    //  modificación de las
           // características para permitir escritura

      pSection->Characteristics = pSection-
       >Characteristics |  0x80000000;

 Section = *pSection;
 return Section;
}
```

Como se puede comprobar, el último parámetro de la función permite modificar sus características. Ello facilitará en el futuro utilizar esta función para localizar información cómodamente sobre la sección.

La función se puede invocar de la siguiente manera:

```
DWORD AdrOfSecTable;
for (DWORD i = 0; i < SecNum; i++)
```

```
{
 AdrOfSecTable = (DWORD)pImageNT+sizeof
  IMAGE_NT_HEADERS+i*(sizeof IMAGE_SECTION_HEADER);
                                          // puntero a la
                                    // tabla de secciones
 *pSection =
  RvaToSection(pImageNT,pMem,AdrOfSecTable,TRUE);
}
```

Desgraciadamente, la función `ImageRvaToSection`, que obtiene información de la sección en el puntero a la estructura `IMAGE_SECTION_HEADER`, no funciona muy bien ya que por alguna razón no puede procesar secciones no estándar (como las que contienen ceros en `PointerToRawData` o `SizeOfRawData`). Para evitar problemas y en caso de error, el valor se puede obtener manualmente mediante la función `memcpy`.

En este caso, la utilización de la función `ImageRvaToSection` resulta lógicamente redundante, aquí se incluye para mostrar todas las opciones. Esta función se podrá utilizar de tarde en tarde para obtener información sobre una sección fácilmente.

Ya se puede utilizar la información sobre la última sección del fichero para calcular los parámetros de la nueva sección:

```
unsigned char Name1 [8] = ".new";
DWORD *VirtualAddress = new DWORD;
DWORD *VirtualSize = new DWORD;
DWORD *SizeOfRawData = new DWORD;
DWORD *PointerToRawData = new DWORD;
DWORD *Characteristics = new DWORD;

*VirtualAddress = PEAlign(pSection-
 >VirtualAddress+pSection->Misc.VirtualSize,pImageNT-
 >OptionalHeader.SectionAlignment);
*VirtualSize = ..(próxima explicación);
*SizeOfRawData = ..(próxima explicación);
*Characteristics = 0xE00000E0;
*PointerToRawData = pSection->PointerToRawData+pSection-
 >SizeOfRawData;
```

La función `PEAlign` obtiene el primer parámetro redondeado al múltiplo más cercano del segundo parámetro. Queda definida de la siguiente manera:

```
DWORD CPEcoderDlg::PEAlign (DWORD Num,DWORD AlignTo)
{
 while(Num % AlignTo != 0)
```

```
{
        Num++;
}
return Num;
}
```

Se utilizará esta función para alinear la sección en memoria. En aras a la sencillez, olvídese que esta función habrá de emplearse en el cálculo de `SizeOfRawData` (con alineamiento de `FileAlignment`). Se añadirá esta función una vez que se haya calculado el valor correcto de `SizeOfRawData`.

Para asegurar que el fichero permanecerá operativo, habrá que modificar ciertos valores de la estructura `IMAGE_NT_HEADERS`: aumentar `SizeOfImage`, `SizeOfCode` y `SizeOfInitializedData` según el tamaño de los datos nuevos, y aumentar el valor en uno de `NumberOfSections` por haber añadido la nueva sección. El código resultante será:

```
pImageNT->FileHeader.NumberOfSections += 1;
pImageNT->OptionalHeader.SizeOfImage += *SizeOfRawData;
pImageNT->OptionalHeader.SizeOfCode += *SizeOfRawData;
pImageNT->OptionalHeader.SizeOfInitializedData +=
*SizeOfRawData;
```

Para añadir la nueva sección con éxito, sólo resta almacenar el contenido de la memoria en el fichero. Si el lector no supiera cómo llevar esto a cabo, podrá encontrar el código fuente completo y el codificador PE ya terminado con comentarios al final de este capítulo.

Redirección de los datos

Tras haber creado una nueva sección llega el momento ahora de definirle código y de dirigirle el flujo de datos. Para empezar, se incluirá código sin ningún objetivo en especial en la sección, por ejemplo un par de instrucciones NOP, y luego se sustituirá con el código de control real. Posteriormente se describirá cómo crear el código de control.

Figura 7-8. La nueva sección incluida ya en el fichero

A la hora de redirigir las instrucciones procesadas por el fichero, constituye un paso esencial sustituir el valor `Program Entry Point` de la estructura `IMAGE_NT_HEADERS` con la RVA de los datos necesarios. Así se explica que tras haber procesado los datos pertinentes, sea necesario volver al código original para cerciorarse que también se procesa (suponiendo que deba ser procesado). Para ello y por el momento, bastará con la instrucción JMP. Se profundizará más en este asunto en las secciones posteriores del capítulo.

Se muestra a continuación un esquema de la operación recién descrita:

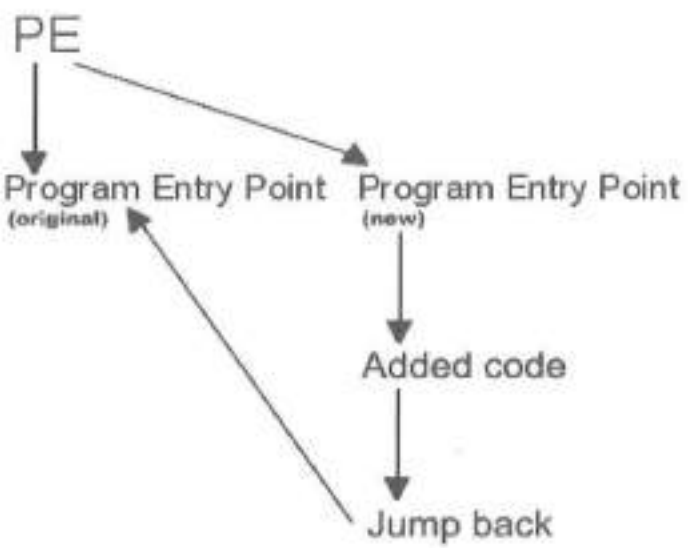

Figura 7-9. Nuevo flujo del Program Entry Point

Seguidamente se muestra el código para redirigir `Program Entry Point`, añadido mediante la función `AddSection`. Basta tan sólo con sustituir la RVA del `Program Entry Point` original con la RVA del comienzo de la sección nueva donde se ubicará el código, esto es, según el valor de `VirtualAddress`.

```
pImageNT->OptionalHeader.AddressOfEntryPoint =
*VirtualAddress;
```

Inclusión de código en una sección nueva

Tal vez la mejor solución para incluir código en una sección nueva del fichero consista en crear otra función en el codificador, en el sitio en el que se guardará el código de control del codificador y donde se definirán las variables globales necesarias para incluirlo.

Esta función puede denominarse, por ejemplo, AddedCode. Está diseñada para que siempre que se ejecute sólo defina las variables (por sencillez se utilizarán variables globales de doble palabra sin explicar cómo se definen) con los datos necesarios, sin que llegue a procesar el código de control del codificador. La invocación a esta función se realizará desde el siguiente sitio:

```
...
AddedCode();    //  carga de variables globales
DWORD FileSize = GetFileSize(File,NULL);
DWORD Size = FileSize+CODE_SIZE+0x1000 /*Buffer
                                    (FileAlignment...)*/;
BYTE *pMem = new BYTE [Size];    //  asignación de
                                 // memoria
```

La definición de la función AddedCode:

```
void __stdcall AddedCode()    //  convención para
                              // invocar __stdcall
{
__asm
{     //  definición de valores en variables globales
 mov eax,offset CodeEnd
 sub eax,offset CodeStart
 mov CODE_SIZE,eax
 mov eax,offset CodeStart
 mov CODE_START,eax
 jmp CodeEnd    // al ejecutar esta función no debe
                //  procesarse el código de control

}
/*** Código de control para incluir en el fichero ***/
CodeStart:
 __asm
 {
       ...
       nop
       nop
       ...
 }
CodeEnd:;
}
```

Como el lector ya habrá anticipado, se puede utilizar a continuación la variable global CODE_SIZE (el tamaño real del código de control) a la hora de asignar la memoria

y calcular el valor real de SizeOfRawData y de VirtualSize para la sección nueva (véase la función AddSection):

```
*SizeOfRawData = PEAlign(CODE_SIZE,pImageNT-
 >OptionalHeader.FileAlignment);
*VirtualSize = CODE_SIZE;
```

La misma función empleada para crear la nueva sección, esto es, AddSection, añadirá el código en memoria. Su definición:

```
memcpy((VOID*)(pMem+*PointerToRawData),(VOID*)(CODE_STAR
 T),CODE_SIZE);
```

Queda destacar el hecho de que en circunstancias normales la suma de SizeOfRawData y de VirtualSize de la última sección debería ser equivalente al tamaño total del fichero entero. Situación que no sucede siempre. Algunos ficheros no tienen que cumplir esta condición y aparentan ocupar más tamaño del que realmente tienen. Por ello, resulta preferible emplear la suma de SizeOfRawData y de VirtualSize (si fuera posible) en vez del tamaño total del fichero. Así se ahorrarán problemas.

Bifurcaciones y variables

Existen varias maneras de volver a los datos originales, o para ser precisos, al Program Entry Point original. Seguidamente se ejemplificará este paso con la opción más sencilla: mediante instrucciones JMP. Por supuesto, el código original podrá invocarse con la instrucción CALL desde el código de control y bifurcar a aquél de nuevo. El número de posibilidades es ilimitado. Ahora bien, recuérdese que, como aspectos críticos de la protección, toda la seguridad del codificador PE radica en el método de guardar y bifurcar al Program Entry Point original. Normalmente, el primer paso de un cracker consiste en buscar el Program Entry Point original para descodificarlo manualmente. Si además se cometiese algún error en la protección (incorrecta comprobación de puntos de corte, ausencia de comprobación de la integridad de los datos, no protegerse contra el volcado de datos, etc.), el cracker dispondrá de todos los recursos para poder descodificar el fichero manualmente.

Resulta preciso almacenar el valor de la bifurcación al Program Entry Point original en algún sitio antes de sobrescribirlo con un valor nuevo. Aquí es donde las variables juegan su papel, o para ser más concretos, el espacio de almacenamiento que el código de control del codificador PE necesita para trabajar. Pronto se observará que el espacio de almacenamiento que el código de control necesita inmediatamente desde su arranque (a este grupo de variables se le denominará variables de inicialización) no constituye el único problema que ha de resolverse a este respecto. A medida que el código

de control se hace más complicado también será necesario más espacio de almacenamiento temporal al que acceder fácil y rápidamente.

Como suele suceder, hay muchas formas de resolver estos problemas. Las variables de inicialización se pueden almacenar directamente en el código de control. Sin embargo, los valores así almacenados resulta fácilmente visibles y la seguridad, nula por tanto. El método más utilizado y bastante mejor para la inicialización y también para las variables temporales, consiste en reservar cierto espacio para ellas en el código de control (con frecuencia, donde éste finalice) y cargarlas desde ahí. Aunque existan muchos otros métodos de almacenamiento para las variables temporales, éste posiblemente resulte el menos complicado para poder luego acceder a ellas.

Si bien desde el punto de vista de la seguridad resulte mejor almacenar las variables en distintos sitios, ha de considerarse que ello acarrea la necesidad de otro código de control que, de todos modos, resultará visible a un depurador. Por lo tanto, resulta preferible almacenar las variables en un solo sitio, cargarlas cómodamente desde ahí y centrarse en un buen mecanismo que las proteja, por ejemplo, codificándolas. Lo que debería aplicarse especialmente a la protección del valor del `Program Entry Point` original.

El valor de `ImageBase` también resulta aquí esencial. El `Program Entry Point` es una RVA a la que debe sumarse el valor de `ImageBase` para obtener la VA real de la bifurcación. No puede almacenarse únicamente el valor final de la suma de estos dos valores puesto que el valor de `ImageBase` puede cambiar en caso de una asignación de memoria fallida, y así dicha suma sería incorrecta. Ese valor debe calcularse mientras el programa esté ejecutándose.

Es cierto que esta afirmación no se puede aplicar en el caso de ficheros EXE, ya que no contienen reubicaciones y por tanto colapsarían si no realizasen la asignación de memoria en la dirección del valor de `ImageBase`. Ahora bien, existen muchos ficheros EXE que sí contienen reubicaciones (por ejemplo, los programas escritos en Delphi); también es importante en el caso de las librerías DLL.

Por motivos pedagógicos en el siguiente ejemplo no se empleará ninguna codificación y ningún otro método de protección de variables adicional. ¡Que no se piense, sin embargo, en hacer lo mismo con un codificador PE real!

Las variables se almacenarán al final del código de control; una variable global, VAR_STAT, marcará el principio de su posición y obtendrá sus valores como en los casos anteriores.

```
void __stdcall AddedCode()    // convención para invocar
                              // __stdcall
```

```
{
 __asm
 {   //  valores de las variables globales
  mov eax,offset CodeEnd
  sub eax,offset CodeStart
  mov CODE_SIZE,eax
  mov eax,offset CodeStart
  mov CODE_START,eax
  mov eax,offset VariablesStart   //  posición de las
                                  // variables
  sub eax,offset CodeStart
  mov VAR_START,eax

  jmp CodeEnd     //  al invocar esta función el código
                  //de control no debe procesarse
 }

/***Código de control para incluir en el fichero***/
CodeStart:
 __asm
 {
  ...
  nop
  nop
  ...

VariablesStart:
/*******Initiation variables********/
NewEntryPointRVA:    //  RVA del nuevo Program Entry
                                                    //Point
  __emit 0        //  se utilizan caracteres cero por
                 // exigirlo la asignación de espacio,
                // serán sustituidos con sus valores
               // precisos
  __emit 0
  __emit 0
  __emit 0

OrgEntryPointRVA:    //  RVA del Program Entry Point
                    //original
  __emit 0
  __emit 0
  __emit 0
  __emit 0

/**Los valores de estas variables se obtendrán al
ejecutarse el programa**/
```

```
ImageBase:
  __emit 0
  __emit 0
  __emit 0
  __emit 0
}

CodeEnd:;
}
```

Obsérvese que el valor de `ImageBase` ha de calcularse en el momento de ejecutar el programa. Para hallar su valor debe procederse de la siguiente manera: una vez guardada la RVA del nuevo Program Entry Point, calcúlese su VA, que podrá determinarse fácilmente a partir del código de control. El resultado será la `ImageBase` actual.

Ahora que ya se dispone de espacio para las variables, podrán obtenerse sus valores. Para simplificar el almacenamiento de las variables de inicialización con sus respectivas direcciones, se utilizará un campo en cuyos ítems se ubicarán los valores de las variables en el mismo orden en el que se encuentran en el código de control (las variables temporales se situarán detrás puesto que se guardarán cuando el programa esté ejecutándose). Este campo se puede ampliar fácilmente para más variables; bastará con un mandato para almacenarlas en el código de control.

A continuación se muestra el código referido:

```
static DWORD VAR [x];
...
/*******Función AddSection*******/
*VirtualAddress = PEAlign(pSection-
 >VirtualAddress+pSection->Misc.VirtualSize,pImageNT-
 >OptionalHeader.SectionAlignment);
*VirtualSize = CODE_SIZE;
*SizeOfRawData = PEAlign(CODE_SIZE,pImageNT-
 >OptionalHeader.FileAlignment);
*Characteristics = 0xE00000E0;
*PointerToRawData = pSection->PointerToRawData+pSection-
 >SizeOfRawData;

/*******Campo con las variables*******/
VAR [0] = *VirtualAddress;
VAR [1] = pImageNT->OptionalHeader.AddressOfEntryPoint;
...

/***Inclusión de las variables en el código de control***/
memcpy((VOID*)(pMem+*PointerToRawData+VAR_START),(VOID*)
 VAR,sizeof VAR);
```

El lector podrá, por supuesto, elegir un procedimiento distinto para almacenar las variables.

Otro asunto por resolver es el de los métodos de acceso a las variables del código de control. El código de control que se muestra a continuación responde a todas estas cuestiones:

```
push ecx
push edx
push esi
push edi
push ebp

call CallMe

CallMe:
pop ebp
sub ebp,offset CallMe

mov ebx,offset CodeStart
add ebx,ebp    //  EBX = VA del nuevo Program Entry
              //Point
sub ebx,[ebp+NewEntryPointRVA]    //  EBX = ImageBase
mov [ebp+ImageBase],ebx    //  guardando el valor de
                           // ImageBase value en una
                           // variable
mov eax,[ebp+OrgEntryPointRVA]    //  EAX = RVA del
                          // Program Entry Point original
add eax,ebx    //  EAX = EAX+EBX = VA del Program Entry
               // Point original
pop ebp
pop edi
pop esi
pop edx
pop ecx
pop ebx
jmp eax    //  bifurcación a los datos originales
```

El primer paso consiste en almacenar los registros específicos (aunque en este punto parezca innecesario guardar ciertos registros, sí que serán de utilidad en el futuro). Seguidamente, la instrucción CALL invoca la instrucción siguiente. Puesto que CALL guarda la dirección de la próxima instrucción en una pila (para poder devolver el control fácilmente), la siguiente instrucción POP EBP almacenará su propia dirección en el

registro EBP. Ahora bien, la próxima instrucción, SUB EBP con desplazamiento CallMe, deberá contener siempre cero por estar obteniendo dos valores equivalentes. ¿Debe ser así? Dependerá de lo que se haga con el código. Si se ejecutara tal cual, el resultado será cero. Pero si se añadiera a otro fichero, se copiará al código del programa ya traducido y así el valor de desplazamiento CallMe será sustituido por un valor numérico con la dirección en el fichero fuente. Los valores obtenidos de esta manera serán, por tanto, distintos y la diferencia resultante, cierto valor de desplazamiento de la posición de las variables entre el fichero fuente y el de destino. El valor de la posición de la variable en el fichero fuente original se sumará a esta diferencia, el resultado constituirá la dirección real de la variable. En pocas palabras: es necesario acordarse de que la posición de los datos no es la misma en el fichero fuente de donde se copia el código de control (codificador PE) que en el fichero de destino. Es preciso primero calcular este desplazamiento y emplearlo para hallar la posición correcta de las variables.

Tras haber calculado el valor de `ImageBase` y cargado la variable `OrgEntryPointRVA`, ambos valores se suman, su resultado será la VA real del `Program Entry Point` original. El resultado se guarda en el registro EAX, los demás registros se redefinen. Resulta innecesario a estas alturas explicar la misión de la instrucción JMP EAX (la bifurcación deseada para volver a los datos originales).

Conviene recordar de nuevo que no debe presentarse el valor inicial del `Program Entry Point` volviendo simplemente a los datos originales: debe codificarse antes, o quizás al lector se le ocurra alguna idea más original. El lector que haya leído el libro desde su principio, ya conoce varios métodos para mover el programa de un sitio a otro disimuladamente —SEH, SMC (ningún buen algoritmo de protección puede prescindir de estas técnicas), etc.—.

Funciones importadas

El código insertado en un programa normalmente emplea cierto número de funciones API. Ahora bien, no se debe confiar en que el código del programa también utilice estas mismas funciones y que asimismo queden disponibles para el código de control. Existen varias maneras de resolver este problema. Se pueden añadir las funciones a la tabla de importaciones (o, con más precisión, desplazarlas a un sitio diferente en el fichero con suficiente espacio), o bien añadirlas a una tabla de importaciones propia e importarlas manualmente, lo que las cargará de la tabla original en el código de control.

Aunque la primera opción sea algo más sencilla, su aplicación para la antipiratería resulta imposible. Ya se ha reincidido varias veces en lo provechoso que resulta la información sobre funciones importadas para un cracker potencial. Es muchísimo mejor añadir una tabla de importaciones nueva al fichero. Ahora bien, esta tabla no deberá contener (por los motivos de seguridad anteriormente mencionados) la lista de todas las funciones API utilizadas por el código de control. Sólo contendrá dos funciones API —`GetProcAddress` y `LoadLibraryA`— de la librería kernel32.dll, útiles para localizar una dirección de cualquier

función API (en otras palabras: para invocar cualquier función API). El código de control invocará estas direcciones en vez de recurrir directamente al nombre de las funciones.

Al añadir la tabla nueva de importaciones al fichero y modificar los valores pertinentes para asegurar que señalan a la tabla de importaciones nueva, el cargador PE no procesará, por supuesto, la tabla original. Será el código del programa, que quedará incorporado al código de control, quien tenga que resolver este problema. A este código del programa se le identificará como importador de funciones.

A pesar de estas consideraciones, la tabla original de importaciones sigue teniendo su importancia. De no modificarse, supondría un agujero en la seguridad.

Entre las funciones más importantes de un codificador PE figuran las de procesar correctamente las importaciones y proteger la tabla original de importaciones. Téngase en cuenta que un cracker potencial nunca podrá ejecutar un fichero descodificado manualmente sin la tabla de importaciones original: su protección constituye un aspecto de la seguridad que debe asumirse con seriedad. Si la tabla original de importaciones quedase sin modificar, el posible cracker, tras descodificar el fichero, no necesitará ni reconstruirlo siquiera.

Un método más inteligente de codificación solventaría este problema (si sencillamente toda la tabla se codifica para luego descodificarla, cualquier cracker podría guardarla ya descodificada). Otra alternativa consiste en desplazar partes importantes de la tabla a ubicaciones diferentes y procesarlas desde allí. Al cracker le costará más reconstruir la tabla original y, lo que es más importante aún, con este método se obstaculiza significativamente el depurado al no resultar visibles los nombres de las funciones.

Toda atención resulta escasa a la hora de procesar y proteger la tabla original de importaciones, ninguna solución llegará a ser demasiado radical en estos casos. No debe olvidarse que los crackers están acostumbrados a desenvolverse en situaciones extremas.

No se concederá más atención a la protección de la tabla original de importaciones. El objetivo primordial de este capítulo consiste en mostrar aplicaciones de la tabla nueva de importaciones y de la importación de funciones para asegurar que el fichero permanece operativo. Queda al arbitrio del lector elegir alguna de las varias soluciones para proteger la tabla de importaciones que hasta aquí se han ofrecido.

CREACIÓN DE UNA TABLA DE IMPORTACIONES

En primer lugar se estudiará cómo crear una tabla nueva de importaciones. Con tal propósito, se definirá una función denominada `AssembleIT`, cuyo objetivo será crear la tabla de importaciones tras el código añadido. Al añadir aquí progresivamente más datos, será necesario introducir otra variable global denominada `OTHER_SIZE`, que albergará los tamaños de estos datos —y que por el momento— será igual al tamaño de la tabla de

importaciones (la variable irá aumentando gradualmente). Nunca se olvide añadir este valor a la hora de asignar memoria.

Llega el momento de centrarse en la construcción de la tabla de importaciones. Nuestra tabla importará sólo de una única librería —kernel32.dll, por lo que contendrá dos estructuras `IMAGE_IMPORT_DESCRIPTOR`— una con la librería y otra finalizando con caracteres nulos. También incluirá dos estructuras `IMAGE_THUNK_DATA` y una de terminación, más dos estructuras `IMAGE_IMPORT_BY_NAME` por tener que importar dos funciones API `GetProcaddress` y `LoadLibraryA`. Todo ello supone, junto con los nombres de las funciones API y la librería DLL, 61h=97 bytes en total (no deben olvidarse los caracteres nulos de finalización detrás cada secuencia de caracteres).

A continuación se muestra un ejemplo de este tipo de tabla (comienza en la posición de memoria 00405000):

```
00405000: 00 00 00 00   00 00 00 00   00 00 00 00   54 50 00 00                TP
00405010: 28 50 00 00   00 00 00 00   00 00 00 00   00 00 00 00  (P
00405020: 00 00 00 00   00 00 00 00   34 50 00 00   42 50 00 00                4P BP
00405030: 00 00 00 00   00 00 40 8F   61 64 64 72   42 72 61 72
LoadLibrar
00405040: 79 41 00 00   00 47 65 74   50 72 6F 63   41 64 64 72  yA GetProcAddr
00405050: 65 73 73 00   6B 65 72 6E   65 6C 33 32   2E 64 6C 6C  ess kernel32.dll
00405060: 00
```

Las primeras cinco dobles palabras constituyen la estructura `IMAGE_IMPORT_DESCRIPTOR`, relativa a la librería DLL desde la que se realiza la importación. No debe darse ningún valor a las tres primeras dobles palabras de la estructura, esto es, `OriginalFirstThunk`, `TimeDateStamp`, y `ForwarderChain`. Sus valores serán nulos. Sin embargo, sí que habrá que definir valores a los ítems `Name` (un puntero RVA a los nombres de la librería) y `FirstThunk` (puntero RVA a la estructura `IMAGE_THUNK_DATA`). Le siguen la estructura final `IMAGE_IMPORT_DESCRIPTOR` con caracteres nulos y tres estructuras `IMAGE_THUNK_DATA`. Las primeras dos estructuras contienen un puntero RVA a las estructuras `IMAGE_IMPORT_BY_NAME` de las funciones LoadLibraryA y GetProcAddress. La tercera es una estructura final conteniendo, de nuevo, caracteres nulos. Luego quedan dos estructuras `IMAGE_IMPORT_BY_NAME` de estas funciones; el ítem `Hint` (primeros dos bytes) no ha de cobrar ningún valor. Debe colocarse un carácter nulo final tras los nombres de las funciones importadas. En último lugar figura el nombre de la librería con un carácter nulo final.

Así quedaría la función `AssembleIT` finalmente:

```
void CPEcoderDlg::AssembleIT(BYTE *pMem,DWORD
 NewSectionRAW,DWORD NewSecti-onVA,PIMAGE_NT_HEADERS
 pImageNT)
{
 DWORD *Name = new DWORD;
 DWORD *FirstThunk = new DWORD;
 DWORD *ThunkLoadLibrary = new DWORD;
```

```
DWORD *ThunkGetProcAddress = new DWORD;
NewSectionRAW += CODE_SIZE;   // inserción de la
                    // tabla tras el código de control
*Name = NewSectionVA+2*sizeof
 IMAGE_IMPORT_DESCRIPTOR+3*sizeof
 IMAGE_THUNK_DATA+12+1+14+1+2*sizeof WORD;
        // Name = Puntero RVA al nombre de la librería
*FirstThunk = NewSectionVA+2*sizeof
 IMAGE_IMPORT_DESCRIPTOR;
                              // FirstThunk = punter
                              // RVA a la estructura
                              // IMAGE_THUNK_DATA
memcpy((VOID*)(pMem+NewSectionRAW+12),Name,sizeof
 DWORD);   // Name

memcpy((VOID*)(pMem+NewSectionRAW+16),FirstThunk,
 sizeof DWORD);   // FirstThunk

*ThunkLoadLibrary = NewSectionVA+2*sizeof
 IMAGE_IMPORT_DESCRIPTOR+3*sizeof IMAGE_THUNK_DATA;
                        // ThunkLoadLibrary = valor de la
                  //estructura IMAGE_THUNK_DATA          //
                              para la función LoadLibraryA

*ThunkGetProcAddress = NewSectionVA+2*sizeof
 IMAGE_IMPORT_DESCRIPTOR+3*sizeof
 IMAGE_THUNK_DATA+sizeof WORD+12+1;

memcpy((VOID*)(pMem+NewSectionRAW+2*sizeof
 IMAGE_IMPORT_DESCRIPTOR),(VOID*)ThunkLoadLibrary,sizeo
 f IMAGE_THUNK_DATA);     // ThunkLoadLibrary

memcpy((VOID*)(pMem+NewSectionRAW+2*sizeof
 IMAGE_IMPORT_DESCRIPTOR+sizeof
 IMAGE_THUNK_DATA),(VOID*)ThunkGetProcAddress,sizeof
 IMAGE_THUNK_DATA);
//  ThunkGetProcAddress
DWORD Address = NewSectionRAW+2*sizeof
 IMAGE_IMPORT_DESCRIPTOR+3*sizeof
 IMAGE_THUNK_DATA+sizeof WORD;
pMem [Address] = 0x4C;   //  L
pMem [Address +1] = 0x6F;   //  o
pMem [Address +2] = 0x61;   //  a
pMem [Address +3] = 0x64;   //  d
pMem [Address +4] = 0x4C;   //  L
pMem [Address +5] = 0x69;   //  i
pMem [Address +6] = 0x62;   //  b
```

```
pMem [Address +7] = 0x72;    //  r
pMem [Address +8] = 0x61;    //  a
pMem [Address +9] = 0x72;    //  r
pMem [Address +10] = 0x79;   //  y
pMem [Address +11] = 0x41;   //  A

Address = NewSectionRAW+2*sizeof
 IMAGE_IMPORT_DESCRIPTOR+3*sizeof
 IMAGE_THUNK_DATA+2*sizeof WORD+12+1;
pMem [Address] = 0x47;       //  G
pMem [Address+1] = 0x65;     //  e
pMem [Address+2] = 0x74;     //  t
pMem [Address+3] = 0x50;     //  P
pMem [Address+4] = 0x72;     //  r
pMem [Address+5] = 0x6F;     //  o
pMem [Address+6] = 0x63;     //  c
pMem [Address+7] = 0x41;     //  A
pMem [Address+8] = 0x64;     //  d
pMem [Address+9] = 0x64;     //  d
pMem [Address+10] = 0x72;    //  r
pMem [Address+11] = 0x65;    //  e
pMem [Address+12] = 0x73;    //  s
pMem [Address+13] = 0x73;    //  s

Address =NewSectionRAW+2*sizeof
 IMAGE_IMPORT_DESCRIPTOR+3*sizeof
 IMAGE_THUNK_DATA+2*sizeof WORD+12+1+14+1;

pMem [Address] = 0x6B;       //  k
pMem [Address+1] = 0x65;     //  e
pMem [Address+2] = 0x72;     //  r
pMem [Address+3] = 0x6E;     //  n
pMem [Address+4] = 0x65;     //  e
pMem [Address+5] = 0x6C;     //  l
pMem [Address+6] = 0x33;     //  3
pMem [Address+7] = 0x32;     //  2
pMem [Address+8] = 0x2E;     //  .
pMem [Address+9] = 0x64;     //  d
pMem [Address+10] = 0x6C;    //  l
pMem [Address+11] = 0x6C;    //  l

delete ThunkGetProcAddress;
delete ThunkLoadLibrary;
delete FirstThunk;
delete Name;

}
```

pMem es el puntero a la memoria asignada, NewSectionRaw es un PointerToRawData a la nueva sección añadida, NewSectionVA es su VirtualAddress y pImageNT, un puntero a la estructura IMAGE_NT_HEADERS.

Con este ejemplo, el lector puede, por supuesto, sustituir los cálculos de las posiciones de las partes individuales de las estructuras con los valores ya definidos por saber ya el tamaño exacto de las estructuras individuales y sus partes. Ahora bien, de esa manera no queda ilustrada la forma de realizar los cálculos individuales. Se emplea a continuación el primer valor calculado, Name (puntero RVA al nombre de la librería) como ejemplo:

```
*Name = NewSectionVA+2*tamaño de
IMAGE_IMPORT_DESCRIPTOR+3*tamaño de
IMAGE_THUNK_DATA+12+1+14+1+2*tamaño de  WORD;
2*tamaño de  IMAGE_IMPORT_DESCRIPTOR = 2*5*DWORD = 40 bytes
3*tamaño de  IMAGE_THUNK_DATA = 3*DWORD = 12 bytes
12+1 - secuencia LoadLibraryA + caracter nulo de terminación
- ítem Nombre de la primera estructura IMAGE_IMPORT_BY_NAME
14+1 - secuencia GetProcAddress + caracter nulo de
terminación - ítem Nombre de la segunda estructura
IMAGE_IMPORT_BY_NAME
2*tamaño de  WORD = 2*2 bytes = 4 bytes - tamaño de los
valores de Hint para ambas estructuras IMAGE_IMPORT_BY_NAME
```

84 bytes en total. Por lo tanto, el cálculo del valor de Name puede también escribirse de la manera siguiente:

```
*Name =NewSectionVA+84;
```

PROCESO DE UNA TABLA DE IMPORTACIONES ORIGINAL

Tras haber creado con éxito una tabla nueva de importaciones, se prestará atención al importador de funciones, quien se encargará de que se procesen las funciones de la tabla original de importaciones.

La clave de un importador (y de todo código que necesite cualquier función API en el futuro) radica en determinar las direcciones de dos funciones API —GetProcAddress y LoadLibraryA— contenidas en la tabla nueva de importaciones. Al procesar las importaciones, el cargador PE sobrescribirá los valores FirstThunk en nuestra tabla de importaciones y los sustituirá con las direcciones correctas de estas funciones API. Bastará con almacenar estos valores en alguna variable para que puedan luego emplearse. A continuación se muestra el código:

```
mov ebx,[ebp+ImageBase]    //  EBX = ImageBase
mov edx,[ebx+3Ch]
add edx,[ebp+ImageBase]    //  EDX señala a
                           // IMAGE_NT_HEADERS
add edx,80h
mov edx,[edx]    //  EDX = Tabla de importaciones RVA
add edx,[ebp+ImageBase]    //  EDX = ImageBase+ Tabla de
                          // importaciones RVA = IT VA
add edx,16    //  desplazamiento al último ítem de la
              // estructura IID - esto es, FirstThunk
mov ecx,dword ptr [edx]    //  ECX = FirstThunk
add ecx,[ebp+ImageBase]    //  ECX señala a la primera
                          // estructura IMAGE_THUNK_DATA
mov edx,[ecx]    //  EDX = dirección de la función
                 // LoadLibraryA
mov [ebp+_LoadLibrary],edx    //  se guarda _LoadLibrary

add ecx,4    //  desplazamiento a la segunda estructura
             // IMAGE_THUNK_DATA
mov edx,[ecx]    //  EDX = dirección de la función
                 // GetProcAddress
mov [ebp+_GetProcAddress],edx    //  se guarda
                                 // _GetProcAddress
```

Antes el código del importador carga la tabla original de importaciones y su primera estructura IMAGE_IMPORT_DESCRIPTOR.

```
//  Por su nombre y comentarios, sobra describir las dos
nuevas variables introducidas
mov edx,[ebp+OrgIT]          //  EDX = RVA de la tabla
                            // original de importaciones
add edx,[ebp+ImageBase]      //  EDX = VA de IT
mov eax,[edx]                //  EAX = OriginalFirstThunk
add eax,[ebp+ImageBase]      //  guárdense siempre las Vas
                            // para facilitar su manejo
mov [ebp+OrgFirstThunk],eax    //  se guarda
                               // OriginalFirstThunk
add edx,12    //  desplazamiento tras OrgFirstThunk,
              //  TimeDateStamp y ForwarderChain
mov eax,[edx]    //  EAX = Name
add eax,[ebp+ImageBase]
mov [ebp+Name1],eax    //  se guarda VA de Name
add edx,4
mov eax,[edx]    //  EAX = FirstThunk
add eax,[ebp+ImageBase]
mov [ebp+FirstThunk],eax    //  se guarda VA de FirstThunk
```

```
add edx,4
mov [ebp+Buffer],edx    //  se guarda un puntero a la
                        // próxima estructura IID
```

El importador consistirá en un par de bucles. El principal se encargará de procesar todas las estructuras IMAGE_IMPORT_DESCRIPTOR y, por ello, de todas las librerías importadas desde el programa. El bucle finaliza en cuanto alcance la última estructura IMAGE_IMPORT_DESCRIPTOR, cuyo contenido consiste en una secuencia de ceros. Si todo marcha conforme a las expectativas, la RVA FirstThunk será equivalente a cero únicamente en la última estructura (lo que no tiene necesariamente que aplicarse a las otras variables de la estructura).

Con objeto de facilitar el manejo de las variables, se añade el valor de ImageBase (traducción de RVA a VA). Ello obliga necesariamente a comprobar su equivalencia con el valor de ImageBase y no con cero.

```
MainITLoop:
 mov ecx,[ebp+ImageBase]
 cmp [ebp+FirstThunk],ecx    //  ¿VA de FirstThunk =
                             // ImageBase?
 jz MainITEnd    //  en caso afirmativo, se habrán
                 // procesado todas las librerías
```

Se carga en el bucle principal la librería pertinente a partir de la estructura IMAGE_IMPORT_DESCRIPTOR mediante la función API LoadLibraryA. Así se llegará a calcular la ImageBase de la librería.

```
mov ebx,[ebp+Name1]
push ebx    //  nombre de la librería
mov eax,[ebp+_LoadLibrary]
call eax    //  CALL LoadLibraryA
//  test eax,eax
//  jz Error
mov ebx,eax    //  EBX = ImageBase de la librería
```

A continuación se utiliza el valor OriginalFirstThunk para hallar la posición de estructuras IMAGE_THUNK_DATA de la librería correspondiente (almacenada en la variable OrgThunkData). Si OriginalFirstThunk fuera cero, se utilizará en su lugar FirstThunk.

```
mov ecx,[ebp+ImageBase]
cmp [ebp+OrgFirstThunk],ecx    //  ¿VA de OrgFirstThunk
                               // = ImageBase?
jnz Nothing    //  ¿RVA de OrgFirstThunk = 0?
mov eax,[ebp+FirstThunk]     //  en caso afirmativo,
                            // colocar el valor
                         // FirstThunk  en OrgFirstThunk
mov [ebp+OrgFirstThunk],eax
Nothing:
mov ecx,[ebp+OrgFirstThunk]
mov eax,[ecx]
mov [ebp+OrgThunkData],eax    // obtención de
                              // IMAGE_THUNK_DATA
```

Se emplea un bucle anidado para calcular las direcciones de todas las funciones que cada librería concreta importa en el programa (mediante la función API `GetProcAddress`), aquéllas sustituirán las estructuras `IMAGE_THUNK_DATA` según la librería correspondiente. Este bucle se repetirá hasta dar con la última estructura `IMAGE_THUNK_DATA` con ceros.

Al principio del bucle se comprueba si la importación es ordinal:

```
SecondITLoop:
 cmp [ebp+OrgThunkData],0
 jz OrgThunkJMP    //  ¿IMAGE_THUNK_DATA = 0?
 mov eax,[ebp+OrgThunkData]
 and eax,IMAGE_ORDINAL_FLAG32
 cmp eax,0    //  ¿importación ordinal?
jnz Ordinal
```

En cuanto la importación no sea ordinal, se cargará el nombre de la función importada desde la estructura `IMAGE_IMPORT_BY_NAME`, la dirección de esta función en la librería se calculará mediante la función API `GetProcAddress`.

```
mov ecx,[ebp+OrgThunkData]
mov edx,[ebp+ImageBase]
add ecx,2    //  saltar ítem Hint
add ecx,edx
mov edi,ecx    //  EDI señala al nombre de la función
push edi
push ebx  //  EBX = ImageBase de la librería
mov eax,[ebp+_GetProcAddress]
call eax    //  CALL GetProcAddress
//  test eax,eax
//  jz Error
jmp Jump
```

Si la importación fuera ordinal, la constante IMAGE_ORDINAL_FLAG32 (=80000000h) se deducirá a partir de la variable OrgThunkData; y la dirección de esta función en la librería, mediante la función API GetProcAddress.

```
Ordinal:
 mov eax,[ebp+OrgThunkData]
 sub eax,80000000h    //  hallando la constante
 push eax
 push ebx
 mov eax,[ebp+_GetProcAddress]
 call eax //  CALL GetProcAddress
 //  test eax,eax
 //  jz Error
```

Ya procede realizar la función principal de todo el importador, esto es, el mismo procedimiento que el cargador PE empleará al procesar las funciones importadas. Los valores de las estructuras IMAGE_THUNK_DATA quedarán sobrescritos con las direcciones de las funciones calculadas.

```
Jump:
 mov esi,[ebp+FirstThunk]
 mov dword ptr [esi],eax    //  sobrescribir
```

Y de aquí, a otra función importada:

```
mov ecx,[ebp+OrgFirstThunk]    //  carga de los valores
                               // originales
mov edx,[ebp+FirstThunk]
add ecx,4    //  desplazamiento para calcular los
             // valores de FirstThunk y OrgFirstThunk
             // siguientes
add edx,4
mov [ebp+OrgFirstThunk],ecx    //  guardando
mov [ebp+FirstThunk],edx
mov eax,[ecx]    //  cálculo del Nuevo valor de
                 // OrgThunkData
mov [ebp+OrgThunkData],eax    //  esto es, contenido de
                              //la estructura IMAGE_THUNK_DATA
jmp SecondITLoop    // proceso de otra función importada
```

Tras haber procesado las funciones individuales de la librería pertinente, puede procesarse otra librería:

```
OrgThunkJMP:
 mov edx,[ebp+Buffer]    //  carga del puntero a la
                         // estructura IID siguiente
 mov eax,[edx]    //  EAX contiene el primer ítem de la
           // estructura, esto es, OriginalFirstThunk
 add eax,[ebp+ImageBase]    //  EAX = VA de
                            // OriginalFirstThunk
 mov [ebp+OrgFirstThunk],eax    //  guardando VA de
                                // OriginalFirstThunk
 add edx,12              //  desplazamiento al ítem Name
 mov eax,[edx]
 add eax,[ebp+ImageBase]
 mov [ebp+Name1],eax    //  guardando VA de Name
 add edx,4               //  desplazamiento a FirstThunk
 mov eax,[edx]
 add eax,[ebp+ImageBase]
 mov [ebp+FirstThunk],eax    //  guardando VA de
                             // FirstThunk
 add edx,4    //  desplazamiento tras FirstThunk
 mov [ebp+Buffer],edx    //  se guarda el puntero a IID
 jmp MainITLoop    //  librería siguiente

MainITEnd:
```

Si se prefiriera transferir parte de la tabla original de importaciones a un sitio diferente (no resultará necesario recordar la necesidad de sobrescribir los datos de la tabla original con valores cualesquiera), bastará con alterar el código de inicialización del importador para cerciorarse de que carga los datos en la posición correcta y no en la dirección de la tabla original. En este caso, recuérdese asignar memoria suficiente para los ítems `OriginalFirstThunk`, `Name` y `FirstThunk` de todas las estructuras `IMAGE_IMPORT_DESCRIPTOR` de la tabla original que se transfieran. Si bien se puede reorganizar más aún la tabla y no sólo las estructuras `IMAGE_IMPORT_DESCRIPTOR`, apenas aportará efecto apreciable a la seguridad, con lo que sólo generará más trabajo innecesariamente.

USO DE UNA FUNCIÓN IMPORTADA

Por último, se ha añadido un ejemplo de función API (distinta a las funciones `GetProcAddress` y `LoadLibraryA`, incluidas directamente en la tabla de importaciones) en el código de control con objeto de añadirla al fichero. Si bien a la mayoría de los lectores les resulte obvio, un pequeño ejemplo podrá esclarecer aún más el caso.

Al conocer las direcciones de `GetProcAddress` y `LoadLibraryA`, se podrá localizar la dirección de cualquier otra función de otra librería e invocarla fácilmente. El procedimiento resulta prácticamente idéntico al del importador de funciones.

En primer lugar habrá de guardarse en algún punto del código de control el nombre de la función que vaya a invocarse junto con el nombre de la librería que la contiene. En el ejemplo siguiente, tras las variables:

```
Function:
  __emit 0x4D    //  M
  __emit 0x65    //  e
  __emit 0x73    //  s
  __emit 0x73    //  s
  __emit 0x61    //  a
  __emit 0x67    //  g
  __emit 0x65    //  e
  __emit 0x42    //  B
  __emit 0x65    //  e
  __emit 0x65    //  e
  __emit 0x70    //  p
  __emit 0    //  no se olvide el cero final

User32:
  __emit 0x75    //  u
  __emit 0x73    //  s
  __emit 0x65    //  e
  __emit 0x72    //
  __emit 0x33    //  3
  __emit 0x32    //  2
  __emit 0x2E    //  .
  __emit 0x64    //  d
  __emit 0x6C    //  l
  __emit 0x6C    //  l
  __emit 0
```

Se puede apreciar que en este ejemplo se ha elegido una función API bien conocida –`MessageBeep`. A continuación se indica el código que recoge esta función:

```
mov eax,[ebp+_LoadLibrary]
mov ecx,offset User32
add ecx,ebp    //  ECX = dirección del nombre de la
               // librería
push ecx
call eax    //  CALL LoadLibraryA
```

```
mov ebx,eax    //  EBX = user32 ImageBase

mov ecx,offset Function
add ecx,ebp  //ECX = dirección del nombre de la función
mov eax,[ebp+_GetProcAddress]
push ecx
push ebx    //  user32 ImageBase
call eax    //  CALL GetProcAddress

push 40h    //  MB_ICONASTERISK - tipo de sonido
call eax    //  CALL MessageBeep
```

Uno de los típicos sonidos de Windows señalará la corrección del procedimiento.

PROCESO TLS

Muchos programas modernos utilizan TLS (almacenamiento local para hilos, en inglés "Thread Local Storage"). Se emplea en aplicaciones multihilo para almacenar datos de un hilo concreto. En el entorno multitarea, así se asegura, por ejemplo, que dos hilos no escriban en la misma variable global o que puedan intercambiar variables entre ellos, etc.

Un programa que utilice TLS puede identificarse inmediatamente al ser distinto de cero el valor `VirtualAddress` de su estructura `IMAGE_DATA_DIRECTORY [9]` (ítem de la tabla TLS) o porque tenga una sección con el nombre .tls (u otro muy semejante).

Al contemplar las direcciones de los ítems de `TLS Table`, se puede apreciar algo incorrecto: su valor señala fuera de la sección .tls.

Esta disposición difiere ligeramente de la que se aplica, por ejemplo, a las funciones importadas o a las reubicaciones, donde las estructuras apuntadas por el ítem del campo DataDirectory se situaban en una sección especial. El ítem `TLS Table` apunta a la estructura `IMAGE_TLS_DIRECTORY32`, que contiene definiciones de algunos valores importantes en TLS y que quedan fuera de la sección .tls.

```
typedef struct _IMAGE_TLS_DIRECTORY32 {
 DWORD StartAddressOfRawData;
 DWORD EndAddressOfRawData;
 PDWORD AddressOfIndex;
 PIMAGE_TLS_CALLBACK *AddressOfCallBacks;
 DWORD SizeOfZeroFill;
 DWORD Characteristics;
}IMAGE_TLS_DIRECTORY32;
```

El ítem `StartAddressOfRawData` es la VA del comienzo de la sección .tls, `EndAddressOfRawData` es la suma de `StartAddressOfRawData` más `VirtualSize`, esto es, la VA del final de la sección sin alineamiento aplicando el valor `SectionAlignment`.

El valor `SizeOfRawData` de la sección .tls es, con frecuencia, cero, lo que indica que esa sección en particular sólo está definida en la tabla de secciones pero no existe en el fichero físicamente: será el cargador PE quien la cree antes de la ejecución. Así se explica que los valores `StartAddressOfRawData` y `EndAddressOfRawData` sean en realidad VAs y no direcciones materiales.

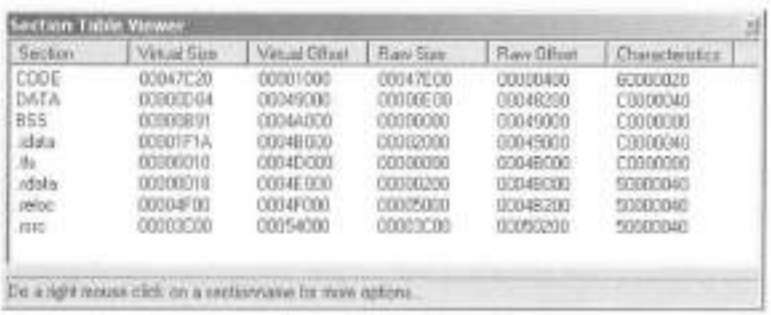

Figura 7-10. Los programas confeccionados con Delphi no sólo contienen la sección .tls

Al trabajar con TLS resulta preciso distinguir entre la propia sección .tls, utilizada como espacio de almacenamiento para datos de hilos específicos, y la estructura definida por TLS.

La sección .tls resulta aquí irrelevante. Sí es importante recordar que no puede codificarse esta sección, ni en el caso de una sección .bss ni en el de ninguna otra con valor cero en `SizeOfRawData`. Causaría un error (además, si el valor `SizeOfRawData` resultase ser cero, no quedaría nada que codificar). Esta sección no se procesará de ningún otro modo.

Mucho más importante resulta la estructura `IMAGE_TLS_DIRECTORY32`. El que esta estructura no se almacene en la sección .tls sino en alguna otra (con frecuencia .rdata), puede causar muchos problemas. Esta estructura podría quedar codificada al codificar alguna otra, lo que provocaría un error en el formato del fichero. En definitiva, esta estructura debe copiarse a algún "sitio seguro" del fichero con la consiguiente modificación de `TLS Table` (variable `VirtualAddress` de la estructura `IMAGE_DATA_DIRECTORY [9]`).

La función que procese TLS puede denominarse, por ejemplo, `ProcessTLS`. A continuación se muestra su definición:

```
void CPEcoderDlg::ProcessTLS(BYTE *pMem,DWORD
VAddress,PIMAGE_NT_HEADERS pImageNT,DWORD FileSize,DWORD
Offset)
{
memcpy((VOID*)(pMem+FileSize+CODE_SIZE),(pMem+Offset),
sizeof IMAGE_TLS_DIRECTORY32);    // se copia la
estructura IMAGE_TLS_DIRECTORY32      // a otro sitio
pImageNT->OptionalHeader.DataDirectory
[9].VirtualAddress = VAddress+CODE_SIZE;
                                                  //  redirección
}
```

El parámetro `pMem` es el puntero a la memoria asignada, `VAddress` es la `VirtualAddress` de la nueva sección, `FileSize` representa el tamaño original del fichero, y, por último, `Offset` es la ubicación material de `IMAGE_TLS_DIRECTORY32` en el fichero.

Esta función copia el contenido de la estructura `IMAGE_TLS_DIRECTORY32` tras el código incluido en la sección nueva (`CODE_SIZE`) para redirigir la variable `VirtualAddress` de la estructura `IMAGE_DATA_DIRECTORY[9]`, que contiene información sobre la ubicación y tamaño de la estructura `IMAGE_TLS_DIRECTORY32`, a la ubicación nueva.

A esta función se la invocará al final de la función `AddSection`:

```
if(pImageNT->OptionalHeader.DataDirectory
[9].VirtualAddress != 0)
{
PIMAGE_SECTION_HEADER Section =
ImageRvaToSection(pImageNT,pMem,
pImageNT->OptionalHeader.DataDirectory
[9].VirtualAddress);
DWORD Offset = pImageNT->OptionalHeader.DataDirectory
[9].VirtualAddress - Section->VirtualAddress+Section-
>PointerToRawData;

ProcessTLS(pMem,*VirtualAddress,pImageNT,*PointerToRawDa
ta,Offset);
}
```

Pueden localizarse los valores `VirtualAddres` y `PointerToRawData` de la sección donde se almacena la estructura `IMAGE_TLS_DIRECTORY32` mediante la función `ImageRvaToSection` para así hallar su posición real en el fichero (y guardar la posición de la variable `Offset`).

CODIFICACIÓN

Elección del algoritmo de codificación

El propósito principal de un codificador PE consiste, por supuesto, en codificar una sección específica del fichero. La elección del algoritmo de codificación y/o descodificación afectará significativamente no sólo la seguridad, sino también la velocidad y la calidad global del codificador PE. A continuación se examinan algunos de los algoritmos de codificación más conocidos.

Algoritmos de codificación comunes

En primer lugar, resulta necesario distinguir entre codificación simétrica y asimétrica. Aquí se aplicarán algoritmos de codificación que podrían considerarse simétricos, esto es, aquellos que utilizan la misma clave tanto para codificar como para descodificar. Los algoritmos asimétricos, por el contrario, emplean claves distintas para codificar y descodificar (por ejemplo, RSA). Una vez generada, la clave se divide en una parte pública y otra privada, y al conjunto denominado pareja de claves.

Los algoritmos de codificación simétricos resultan significativamente más rápidos que los asimétricos y, por lo tanto, más indicados para los codificadores PE.

Por otra parte, algunos algoritmos de codificación asimétricos, como es el caso de RSA, resultan críticos para la codificación de datos. Razón que justifica examinarlos con detalle al final del capítulo, tanto en su vertiente teórica como práctica, y de igual modo, el diseño de algoritmos para calcular sus claves de codificación y descodificación.

Algunos de los algoritmos simétricos profesionales más conocidos se indican a continuación:

IDEA

Este algoritmo destaca especialmente por su velocidad de codificación (mucho más rápido que DES) empleando una longitud de la clave suficientemente larga (128 bits). Ascom-Tech es la propietaria de la patente en la mayoría de los países europeos y en EE.UU.

BlowFish

Bruce Schneir diseñó este rápido algoritmo con la máxima longitud de clave: 448 bits. No está patentado, se pretende que sea de libre circulación, lo que le convierte en una opción excelente para muchos programadores.

CAST

El nombre recoge las iniciales de sus autores: Carlisle Adamsem y Stafford Tavemsem. Aunque muy parecido a BlowFish, desgraciadamente este otro algoritmo no es gratis.

DES

Este estándar de cifrado, desarrollado por IBM, llegó a ser la norma estadounidense de codificación en 1977. La longitud de la clave de este algoritmo es tan sólo de 56 bits, claramente insuficiente en el presente: puede anularse mediante una ataque masivo utilizando, por ejemplo, el programa DES Cracker.

3DES

Resulta prácticamente idéntico al algoritmo DES, si bien cifra los datos tres veces. Duplica la longitud de la clave (112) pero su velocidad disminuye lógicamente en un tercio.

AES

Este algoritmo, vigente desde el 26/05/2002, vino a sustituir, como nuevo estándar del gobierno federal norteamericano, a DES. AES (Estándar de cifrado avanzado, en inglés, "Advanced Encryption Standard") soporta los 256 bits como máxima longitud de clave, la duración estimada de su coeficiente de seguridad llega a los 20 años. En Internet puede encontrarse el código fuente de este algoritmo en diferentes lenguajes de programación.

En lo que a la compresión respecta, el CD adjunto contiene (documentación y ejemplos incluidos) una de las mejores librerías de compresión existentes en la actualidad: aPLib. El único aspecto negativo de esta librería reside en su carácter comercial. Si se pretendiera distribuir software donde se hiciera uso de esta librería, debería antes quedar registrada.

Si el lector deseara crear su propio algoritmo de compresión (en cuyo caso no debe olvidar consultar el epígrafe "Violación del código"), la sección de referencia contendrá un ejemplo de algoritmo de compresión escrito en C++.

Por resultar más pedagógico trabajar con algoritmos de codificación no muy complejos, en los ejemplos siguientes la codificación se llevará a cabo mediante XOR, fácilmente sustituible por cualquier otro algoritmo de compresión. Internet ofrece una gran cantidad de algoritmos de compresión además de información sobre cómo crearlos. La operación lógica XOR (disyunción exclusiva) se comporta conforme se indica a continuación:

A XOR B = C

C XOR B = A

Según se puede observar, dados dos valores, A y B por ejemplo, siendo A los datos que se van a cifrar y B, la clave de cifrado, relacionados mediante un operador XOR, los datos cifrados serán C. Relacionando entonces C con la clave de cifrado B mediante el operador XOR, se obtendrán los datos originales (descifrados). Por tanto, aplicando un simple XOR para cifrar los datos, no existirá diferencia alguna entre el proceso de codificación y descodificación.

A continuación se muestra la tabla de verdad para el operador XOR:

A	B	C = A XOR B
1	1	0
1	0	1
0	1	1
0	0	0

En ensamblador:

Por ejemplo: xor eax,2 <-- EAX = EAX XOR 2

Los algoritmos de codificación basados en la operación lógica XOR normalmente emplean varias tablas con los valores codificados que van modificándose según la clave de cifrado.

Violación del código

Siempre que se elija un procedimiento para generar la clave de descifrado incorrecto o no se utilice ninguno en absoluto (entendiendo por tales aquellos que aplican algoritmos de "cifrado" basados en complejas operaciones matemáticas), se corre el riesgo de que alguien viole el código.

El mejor método para generar la clave de descodificación es el basado en su dependencia del estado de protección de los elementos del código de control. No es infrecuente encontrar protecciones que realizan una comprobación de integridad ("checksum", en inglés) con los datos y luego la emplean como clave de descodificación. Hay muchas otras alternativas, por supuesto, en última instancia dependerá de la creatividad del desarrollador. Ahora bien, en ningún caso debe guardarse la clave de

descodificación directamente en el código de control. Se corre el riesgo de crear un descodificador universal.

Toda la teoría sobre la posible violación del código se basa en el hecho de que si el cracker no cuenta con toda la información necesaria (y que normalmente adquiere utilizando un depurador), no podrá ni siquiera intentar violar el código. A la hora de buscar esta información, intentará anular todos los elementos anti... (antidepuración, anti puntos de corte) del código de control. No parece muy lógico que el paso siguiente sea intentar violar el código de control ya que generaría trabajo extra. El cracker esperará hasta que el código de control realice para él todas las operaciones necesarias (descodificación incluida).

Si los procedimientos aplicados en la codificación y el método de generación de la clave no resultan obvios sin estudiar el código de control, el diseño de éste adquirirá mucha más importancia que el algoritmo de codificación.

La situación varía enormemente al crear un descodificador universal. Resulta mucho más fácil para un cracker escribir un programa que descodifique el fichero mediante una clave de descodificación fija en vez de escribir un programa que ejecute el fichero y guarde el contenido en memoria, donde el fichero se descodifica. Como normalmente se emplea ProcDump para esta operación, deberá dotarse al programa de defensas contra esta herramienta. Un algoritmo de descodificación debe programarse de tal manera que resulte muy difícil sino imposible crear un descodificador universal que descodifique el fichero externamente sin llegarlo a ejecutar realmente. Si a ello se le suma una buena protección frente a ProcDump (o herramientas semejantes) y frente a los métodos que ProcDump emplea (volcado, rastreo), la creación de un descodificador universal se convertirá en una tarea muy difícil.

El anteriormente mencionado codificador PE, PE Shield, constituye un ejemplo perfecto. Este codificador representa un difícil rompecabezas para los crackers por emplear codificación polimórfica y una fuerte protección contra ProcDump y otros descodificadores genéricos.

Los ataques masivos no constituyen tampoco una excepción. Algunas personas se molestan en buscar errores en el algoritmo de descodificación que les permita descodificar los datos mediante ataques masivos durante un tiempo razonable. Ya se puso antes como ejemplo el error en la rutina de descodificación de SafeDisc. R!SC descubrió este error e incorporó un descodificador basado en un ataque masivo que puede utilizarse para descodificar el fichero protegido incluso sin disponer del CD original.

Áreas codificadas y no codificadas

Resultaría muy pueril pensar que la codificación indiscriminada de todas las secciones del fichero no causaría ningún error y que el fichero aún quedara plenamente

operativo. De hecho, sólo se pueden codificar aquellas secciones que contengan datos que no sean necesarios para la inicialización del cargador PE. La situación varía para cada sección: dependerá de quién realice el proceso (el cargador PE o el programa original), de la sección en sí, de los datos que contenga, de cómo se desee codificar los datos, qué parte, etc.

A estas alturas, tras haber examinado los distintos problemas relacionados con el proceso y codificación de algunas secciones, el lector ya se habrá hecho una idea aproximada de esta problemática. A modo de resumen, se podrían señalar las conclusiones siguientes:

Nunca se deben codificar secciones con un valor igual a cero en `PointerToRawData` (por ejemplo, secciones .bss).

En el caso de ficheros EXE, la codificación y la consiguiente necesidad de procesar la sección con la tabla de exportaciones (generalmente .edata) representan un trabajo adicional innecesario. Los ficheros EXE normalmente no contienen tabla de exportaciones y si lo hacen, carece de utilidad para el potencial cracker.

Aunque por la codificación de otras secciones, el programa original sea el encargado de procesar las reubicaciones (normalmente .reloc) para cerciorarse de que no se produce ningún error, también resulta, en la mayoría de las ocasiones, innecesario. Los ficheros EXE tampoco suelen incluirlas y si lo hacen, el posible cambio en la dirección de carga en la mayoría de ficheros de este tipo suele venir causado por algún error crítico. Ya se mencionó todo ello con anterioridad, al explicar el término `ImageBase`.

Las librerías DLL sí deben procesar reubicaciones. En la parte de referencia del libro puede hallarse información suficiente a este respecto así como un algoritmo con dicho fin. Representa un pérdida de tiempo codificar las reubicaciones, carecen de utilidad para un posible cracker.

En lo que atañe al TLS, las secciones con recursos (.rsrc...) también pueden codificarse y procesarse. Resulta más complicado, pero aumenta la seguridad (consúltese el CD adjunto si se desea más información sobre cómo llevarse a cabo).

Ya quedó explicado cómo se codifican y procesan las funciones importadas. Aunque se hayan mencionado los elementos de seguridad correspondientes a la tabla original de importaciones, también se puede ubicar el importador de funciones tras el bucle de descodificación y codificar entonces la sección con la tabla original de importaciones de la misma manera que las demás secciones (por supuesto, dependerá de la protección elegida para la tabla de importaciones original). Así se procederá en el ejemplo posterior.

Hasta aquí lo que respecta a los codificadores PE. Los compresores PE les suele bastar con comprimir procurando procesar la mayor cantidad de datos posible para obtener los mejores resultados de compresión.

Ejemplo de una codificación sencilla con un codificador PE

A continuación se mostrará cómo incluir un algoritmo de codificación muy sencillo al codificador PE. Con objeto de no complicar la aplicación, se utilizará una clave de cifrado fija y un sencillo algoritmo XOR. Debe recordarse una vez más que este tipo de codificación resulta absolutamente inaceptable en un codificador PE y que si aquí se utiliza es sólo por motivos pedagógicos.

La función de codificación es realmente elemental:

```
void CPEcoderDlg::SimpleCrypt(BYTE *pInput,DWORD lSize)
{
for (DWORD lCount = 0; lCount < lSize; lCount++,
pInput++)
   {
     *pInput ^= 0xAB;    //  XOR
   }
}
```

El primer parámetro de la función es un puntero a los datos que se van codificar, y el segundo, su tamaño.

A continuación se crea la función (denominada CryptPE) que codifica todas las secciones pertinentes del fichero:

```
void CPEcoderDlg::CryptPE(PIMAGE_NT_HEADERS
 pImageNT,BYTE *pMem)
{
IMAGE_SECTION_HEADER *pSection = new
 IMAGE_SECTION_HEADER;
DWORD SecNum = pImageNT->FileHeader.NumberOfSections;
                                    // número de secciones
 DWORD AdrOfSecTable;

      for (DWORD i = 0 ;i < SecNum; i++)
 {
       AdrOfSecTable = (DWORD)pImageNT+(sizeof
        IMAGE_NT_HEADERS)+i*(sizeof
        IMAGE_SECTION_HEADER);
                        // puntero a la tabla de secciones
```

```
            *pSection =
             RvaToSection(pImageNT,pMem,AdrOfSecTable);

            if (*(LPDWORD)pSection->Name != 0x77656E2E
             /*".new "*/ &&
            *(LPDWORD)pSection->Name != 0x7273722E /*".rsr
             "*/ &&
            *(LPDWORD)pSection->Name != 0x63727372 /*"rsrc
             "*/ &&
            *(LPDWORD)pSection->Name != 0x6C65722E /*".rel
             "*/ &&
            *(LPDWORD)pSection->Name != 0x6F6C6572 /*"relo
             "*/ &&
            *(LPDWORD)pSection->Name != 0x6164652E /*".eda
             "*/ &&
            *(LPDWORD)pSection->Name != 0x736C742E /*.tls*/
             &&
            pSection->PointerToRawData != 0 && pSection-
             >SizeOfRawData != 0)
            {
                 DWORD Size = (pSection->Misc.VirtualSize
                  <pSection->SizeOfRawData ? pSection-
                  >Misc.VirtualSize :pSection-
                  >SizeOfRawData);
                              // tamaño de los datos cifrados
                 SimpleCrypt(pMem+pSection-
                  >PointerToRawData,Size);
            }
        }
        delete pSection;
    }
```

Durante la secuencia se decide qué secciones se codifican a partir de los cuatro primeros caracteres del nombre de la sección y de un valor distinto a cero en `PointerToRawData` y `SizeOfRawData`.

El nombre de la sección suele reflejar sus características y la necesidad de codificar su contenido. Aunque este método no parezca muy profesional, sí cumple con su misión. La mejor solución exigiría comprobar su contenido (buscando estructuras importantes), y no el nombre de la sección para así poder elegir directamente las secciones que se vayan a codificar.

No se codificarán secciones con nombres tales como .reloc (reubicación), .eadata (funciones exportadas), .tls, .rsrc, (recursos) ni, por supuesto, las recién incluidas con el nombre .new. Puede ignorarse la tabla original de importaciones puesto que el bucle de

descodificación se situará en el código de control, frente al importador de funciones, quien recibirá entonces los datos ya descodificados.

Con objeto de evitar la codificación y descodificación del alineamiento de caracteres nulos en las secciones y evitar posibles problemas posteriormente, el tamaño de los datos codificados vendrá dado por el menor de los valores de `VirtualSize` y `SizeOfRawData`.

Esto, en lo que atañe a la codificación de las secciones. Deberá modificarse el código de control para permitir la descodificación. Para ello, primero se modificará su comienzo (la variable `SecNum` albergará el número de secciones del fichero) para incluir la rutina de descodificación. El algoritmo de descodificación quedará ubicado en la posición siguiente:

```
    /*******Código de control para incluir en el fichero
*******/
    CodeStart:
     __asm
     {
      push ebx
      push ecx
      push edx
      push esi
      push edi
      push ebp

      call CallMe

    CallMe:
      pop ebp
      sub ebp,offset CallMe
      mov ebx,offset CodeStart

      add ebx,ebp    //  EBX =VA del nuevo Program Entry
                     // Point
      sub ebx,[ebp+NewEntryPointRVA]    //  EBX = ImageBase
      mov [ebp+ImageBase],ebx    //  se guarda el valor
                                 // ImageBase en una variable

      mov edx,[ebx+3Ch]
      add edx,[ebp+ImageBase]    //  EDX señala a
                                 // IMAGE_NT_HEADERS
      mov [ebp+Buffer],edx
      add edx,6
      mov bx,[edx]
```

```
mov word ptr [ebp+SecNum],bx    // SecNum = número de
                                // secciones del fichero
add edx,7Ah
mov edx,[edx]    //  EDX = tabla de importaciones RVA
add edx,[ebp+ImageBase]    //  EDX = ImageBase+ tabla
                           // de importaciones RVA

add edx,16
mov ecx,dword ptr [edx]    //  ECX = FirstThunk
add ecx,[ebp+ImageBase]    //  ECX = ECX+ImageBase
mov edx,[ecx]    //  EDX = dirección de la función
                 // LoadLibraryA
mov [ebp+_LoadLibrary],edx    //  guardando en
                              // _LoadLibrary

add ecx,4
mov edx,[ecx]    //  EDX = dirección de la función
                 // GetProcAddress
mov [ebp+_GetProcAddress],edx    //  guardando a
                                 // _GetProcAddress
mov eax,[ebp+EntryPoint]    //  EAX = RVA de Program
                            // Entry Point original
add eax,[ebp+ImageBase]
mov [ebp+OrgEP],eax

/*******Descodificación*******/
...
...

/*******Final del algoritmo de descodficación *******/
mov edx,[ebp+OrgIT]    //  EDX = RVA de la tabla de
                       // importaciones original
add edx,[ebp+ImageBase]    //  EDX = VA de IT
mov eax,[edx]    //  EAX =RVA de OriginalFirstThunk
add eax,[ebp+ImageBase]    //  se guardan Vas para
                           // facilitar su manejo
mov [ebp+OrgFirstThunk],eax    //  se guarda
                               // OriginalFirstThunk
add edx,12    //  desplazamiento tras OrgFirstThunk,
              // TimeDateStamp una ForwarderString
mov eax,[edx]    //  EAX = Name
add eax,[ebp+ImageBase]
mov [ebp+Name1],eax    //  se guarda VA de Name
add edx,4
mov eax,[edx]    //  EAX = FirstThunk
add eax,[ebp+ImageBase]
```

```
mov [ebp+FirstThunk],eax    //  se guarda VA de
                            // FirstThunk
add edx,4
mov [ebp+Buffer],edx    //  se guarda el puntero a la
                        // estructura IID por comodidad

/*******MainITLoop*******/
...
```

Consideremos los elementos individuales del algoritmo de descodificación, no muy diferente del algoritmo de codificación, lo que ahorrará algunas explicaciones. Por emplear un XOR, el algoritmo de codificación y descodificación resultan idénticos.

```
/*******Decoding*******/
xor ecx,ecx    // ECX = 0 - contador de secciones
               // procesadas
mov edi,[ebp+SecNum]
mov eax,[ebp+Buffer]
add eax,0F8h    //  EAX apunta a la primera estructura
                // IMAGE_SECTION_HEADER

MainDecodeLoop:
cmp di,cx    //  ¿se han procesado todas las secciones
             // del fichero?
jz DecodingDone
push edi

/*******¿Debe descodificarse esta sección?*******/
mov ebx,[eax]    //  Name
cmp ebx,77656E2Eh    //  .new
jz NoDecoding
cmp ebx,7273722Eh    //  .rsr
jz NoDecoding
cmp ebx,63727372h    //  rsrc
jz NoDecoding
cmp ebx,6C65722Eh    //  .rel
jz NoDecoding
cmp ebx,6F6C6572h    //  relo
jz NoDecoding
cmp ebx,6164652Eh    //  .eda
jz NoDecoding
cmp ebx,736C742Eh    //  .tls
jz NoDecoding

add eax,8    //  carga de información sobre la sección
   // a partir de la estructura IMAGE_SECTION_HEADER
```

```
mov edi,[eax]    //  VirtualSize
add eax,4
mov ebx,[eax]    //  VirtualAddress
add eax,4
mov edx,[eax]    //  SizeOfRawData
add eax,4
mov esi,[eax]    //  PointerToRawData

cmp esi,0    //  ¿PointerToRawData = 0 ?
jz NoDecoding
cmp edx,0    //  ¿SizeOfRawData = 0 ?
jz NoDecoding

cmp edi,edx
jnl TakeSizeOfRawData    //  ¿VirtualSize <
                                        // SizeOfRawData ?
mov edx,edi    //  en caso afirmativo, se tomará este
               // valor como el tamaño de los datos por
               // descodificar (véase algoritmo de
              // codificación)
TakeSizeOfRawData:
 mov esi,[ebp+ImageBase]
 add ebx,esi    //  EBX = VA de los datos por
               // descodificar
 call SimpleCrypt    //  rutina de descodificación
 jmp SectionDecoded    //  sección descodificada, buscar
                       // siguiente

NoDecoding:
 add eax,14h    //  sección no procesada, necesario
               // omitir su estructura
               // IMAGE_SECTION_HEADER para buscar otra
               // sección(2*ADD EAX,14h)

SectionDecoded:
 add eax,14h    //  omitiendo los restantes ítems de la
              // estructura IMAGE_SECTION_HEADER
 inc cx    //  aumentando el contador de secciones
 pop edi
 jmp MainDecodeLoop

DecodingDone:
```

Los mejores codificadores PE no descodifican las secciones una sola vez y de manera simultánea, así resulta mucho más difícil guardar sus contenidos en disco en formato descodificado. Debe tenerse esto tan en cuenta como el incluir el código de control del codificador PE. Así se evitará su desensamblado.

DISEÑO FINAL DE UN CODIFICADOR PE

Se expone a continuación el código íntegro del codificador PE. Puede resultar útil para quienes tengan dudas sobre algunos de los pasos o no sepan cómo programar aquellas partes no descritas explícitamente. Para facilitar la lectura del código, los comentarios figuran al final de las partes principales.

```
// PEcoderDlg.cpp : fichero con el código
//

#include "stdafx.h"
#include "PEcoder.h"
#include "PEcoderDlg.h"
#include "imagehlp.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////
// CPEcoderDlg dialog

CPEcoderDlg::CPEcoderDlg(CWnd* pParent /*=NULL*/)
  : CDialog(CPEcoderDlg::IDD, pParent)
{
  //{{AFX_DATA_INIT(CPEcoderDlg)
  // NOTA: ClassWizard incluirá aquí el miembro de
  // inicialización
  //}}AFX_DATA_INIT
  m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CPEcoderDlg::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CPEcoderDlg)
  // NOTA: ClassWizard añadirá aquí las llamadas a DDX y
  // DDV
  //}}AFX_DATA_MAP
}
```

```
BEGIN_MESSAGE_MAP(CPEcoderDlg, CDialog)
 //{{AFX_MSG_MAP(CPEcoderDlg)
 ON_WM_PAINT()
 ON_WM_QUERYDRAGICON()
 ON_BN_CLICKED(IDC_OK, OnOk)
 //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////////////////////////////
// manejadores de mensaje CPEcoderDlg

BOOL CPEcoderDlg::OnInitDialog()
{
 CDialog::OnInitDialog();

 SetIcon(m_hIcon, TRUE);        // Definición de icono
                                // grande
 SetIcon(m_hIcon, FALSE);       // Definición de icono
                                // pequeño

 // Añádase aquí código extra de inicialización

 return TRUE;  // TRUE a no ser que se cambie el flujo
               // de control
}

// Si en el cuadro de diálogo se incluye un botón de
// minimización, será necesario añadir el código
// siguiente para dibujar el icono. Con aplicaciones
// MFC que utilicen el modelo documento/ver, la
// infraestructura común se encarga de realizar esta
// operación.

void CPEcoderDlg::OnPaint()
{
 if (IsIconic())
 {
       CPaintDC dc(this); // contexto de dispositivo
                                          // para pintar

       SendMessage(WM_ICONERASEBKGND, (WPARAM)
 dc.GetSafeHdc(), 0);

       //  Centrar icono en el rectángulo del cliente
       int cxIcon = GetSystemMetrics(SM_CXICON);
```

```
        int cyIcon = GetSystemMetrics(SM_CYICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;

        // Dibujar icono
        dc.DrawIcon(x, y, m_hIcon);
    }
    else
    {
        CDialog::OnPaint();
    }
}

HCURSOR CPEcoderDlg::OnQueryDragIcon()
{
    return (HCURSOR) m_hIcon;
}

DWORD CODE_SIZE = 0;
DWORD CODE_START = 0;
DWORD VAR_START = 0;
DWORD OTHER_SIZE = sizeof IMAGE_TLS_DIRECTORY32 + 0x61;

static DWORD VAR[3];

void __stdcall AddedCode()
{
    __asm
    {
        mov eax,offset CodeEnd
        sub eax,offset CodeStart
        mov CODE_SIZE,eax
        mov eax,offset CodeStart
        mov CODE_START,eax
        mov eax,offset VariablesStart
        sub eax,offset CodeStart
        mov VAR_START,eax

        jmp CodeEnd
    }

/****Código de control para incluir en el fichero****/
CodeStart:
    __asm
    {
```

```
        push ebx
        push ecx
        push edx
        push esi
        push edi
        push ebp

        call CallMe

CallMe:
        pop ebp
        sub ebp,offset CallMe

        mov ebx,offset CodeStart
        add ebx,ebp
        sub ebx,[ebp+NewEntryPointRVA]
        mov [ebp+ImageBase],ebx

        mov edx,[ebx+3Ch]
        add edx,[ebp+ImageBase]
        mov [ebp+Buffer],edx
        add edx,6
        mov bx,[edx]
        mov word ptr [ebp+SecNum],bx
        add edx,7Ah
        mov edx,[edx]
        add edx,[ebp+ImageBase]

        add edx,16
        mov ecx,dword ptr [edx]
        add ecx,[ebp+ImageBase]
        mov edx,[ecx]
        mov [ebp+_LoadLibrary],edx

        add ecx,4
        mov edx,[ecx]
        mov [ebp+_GetProcAddress],edx

        mov eax,[ebp+OrgEntryPointRVA]
        add eax,[ebp+ImageBase]
        mov [ebp+OrgEP],eax

        /*******Secciones para descodificar*******/
        xor ecx,ecx
        mov edi,[ebp+SecNum]
        mov eax,[ebp+Buffer]
        add eax,0F8h
```

```
MainDecodeLoop:
        cmp di,cx
        jz DecodingDone

        push edi
        mov ebx,[eax]
        cmp ebx,77656E2Eh
        jz NoDecoding
        cmp ebx,7273722Eh
        jz NoDecoding
        cmp ebx,63727372h
        jz NoDecoding
        cmp ebx,6C65722Eh
        jz NoDecoding
        cmp ebx,6F6C6572h
        jz NoDecoding
        cmp ebx,6164652Eh
        jz NoDecoding
        cmp ebx,736C742Eh
        jz NoDecoding

        add eax,8
        mov edi,[eax]
        add eax,4
        mov ebx,[eax]
        add eax,4
        mov edx,[eax]
        add eax,4
        mov esi,[eax]

        cmp esi,0
        jz SectionDecoded
        cmp edx,0
        jz SectionDecoded

        cmp edi,edx
        jnl TakeSizeOfRawData
        mov edx,edi

TakeSizeOfRawData:
        mov esi,[ebp+ImageBase]
        add ebx,esi
        call SimpleCrypt
        jmp SectionDecoded

NoDecoding:
        add eax,14h
```

```
SectionDecoded:
        add eax,14h
        inc cx
        pop edi
        jmp MainDecodeLoop

DecodingDone:
        mov edx,[ebp+OrgIT]
        add edx,[ebp+ImageBase]
        mov eax,[edx]
        add eax,[ebp+ImageBase]
        mov [ebp+OrgFirstThunk],eax
        add edx,12
        mov eax,[edx]
        add eax,[ebp+ImageBase]
        mov [ebp+Name1],eax
        add edx,4
        mov eax,[edx]
        add eax,[ebp+ImageBase]
        mov [ebp+FirstThunk],eax
        add edx,4
        mov [ebp+Buffer],edx

  /*******Proceso con las funciones importadas*******/
MainITLoop:
        mov ecx,[ebp+ImageBase]
        cmp [ebp+FirstThunk],ecx
        jz MainITEnd

        mov ebx,[ebp+Name1]
        push ebx
        mov eax,[ebp+_LoadLibrary]
        call eax
        mov ebx,eax

        mov ecx,[ebp+ImageBase]
        cmp [ebp+OrgFirstThunk],ecx
        jnz Nothing
        mov eax,[ebp+FirstThunk]
        mov [ebp+OrgFirstThunk],eax

Nothing:
        mov ecx,[ebp+OrgFirstThunk]
        mov eax,[ecx]
        mov [ebp+OrgThunkData],eax
```

```
SecondITLoop:
        cmp [ebp+OrgThunkData],0
        jz OrgThunkJMP
        mov eax,[ebp+OrgThunkData]
        and eax,IMAGE_ORDINAL_FLAG32
        cmp eax,0
        jnz Ordinal

        mov ecx,[ebp+OrgThunkData]
        mov edx,[ebp+ImageBase]
        add edx,2
        add ecx,edx
        mov edi,ecx
        push edi
        push ebx
        mov eax,[ebp+_GetProcAddress]
        call eax
        jmp Jump

Ordinal:
        mov eax,[ebp+OrgThunkData]
        sub eax,80000000h
        push eax
        push ebx
        mov eax,[ebp+_GetProcAddress]
        call eax

Jump:
        mov esi,[ebp+FirstThunk]
        mov dword ptr [esi],eax
        mov ecx,[ebp+OrgFirstThunk]
        mov edx,[ebp+FirstThunk]
        add ecx,4
        add edx,4
        mov [ebp+OrgFirstThunk],ecx
        mov [ebp+FirstThunk],edx
        mov eax,[ecx]
        mov [ebp+OrgThunkData],eax
        jmp SecondITLoop

OrgThunkJMP:
        mov edx,[ebp+Buffer]
        mov eax,[edx]
        add eax,[ebp+ImageBase]
        mov [ebp+OrgFirstThunk],eax
        add edx,12
        mov eax,[edx]
```

```
        add eax,[ebp+ImageBase]
        mov [ebp+Name1],eax
        add edx,4
        mov eax,[edx]
        add eax,[ebp+ImageBase]
        mov [ebp+FirstThunk],eax
        add edx,4
        mov [ebp+Buffer],edx
        jmp MainITLoop

MainITEnd:
 /*******Ejemplo de invocación de función API*******/
        mov eax,[ebp+_LoadLibrary]
        mov ecx,offset User32
        add ecx,ebp
        push ecx
        call eax
        mov ebx,eax

        mov ecx,offset Function
        add ecx,ebp

        mov eax,[ebp+_GetProcAddress]
        push ecx
        push ebx
        call eax

        push 40h
        call eax

        mov eax,[ebp+OrgEP]
        pop ebp
        pop edi
        pop esi
        pop edx
        pop ecx
        pop ebx

  /*******Vuelta al Program Entry Point original*******/
        jmp eax

  /*******Sencillo algoritmo de des/codificación*******/
 SimpleCrypt:
        push ecx
        push eax
        xor ecx,ecx
```

```
    DecodeLoop:
            cmp edx,ecx
            jz DecodeDone
            mov al,[ebx]
            xor al,0ABh
            mov [ebx],al
            inc ebx
            inc ecx
            jmp DecodeLoop

    DecodeDone:
            pop eax
            pop ecx

            ret

            /*******Variables*******/
    VariablesStart:

            /*******Variables de inicialización********/
    NewEntryPointRVA:
            __emit 0
            __emit 0
            __emit 0
            __emit 0

    OrgEntryPointRVA:
            __emit 0
            __emit 0
            __emit 0
            __emit 0

    OrgIT:
            __emit 0
            __emit 0
            __emit 0
            __emit 0

/****Variables empleadas cuando se ejecute el programa****/
    _LoadLibrary:
            __emit 0
            __emit 0
            __emit 0
            __emit 0
```

```
_GetProcAddress:
        __emit 0
        __emit 0
        __emit 0
        __emit 0

ImageBase:
        __emit 0
        __emit 0
        __emit 0
        __emit 0

OrgEP:
        __emit 0
        __emit 0
        __emit 0
        __emit 0

SecNum:
        __emit 0
        __emit 0

OrgFirstThunk:
        __emit 0
        __emit 0
        __emit 0
        __emit 0

FirstThunk:
        __emit 0
        __emit 0
        __emit 0
        __emit 0

Name1:
        __emit 0
        __emit 0
        __emit 0
        __emit 0

OrgThunkData:
        __emit 0
        __emit 0
        __emit 0
        __emit 0
```

```
    Buffer:
            __emit 0
            __emit 0
            __emit 0
            __emit 0

/***Variables necesarias para invocar una función API***/
    Function:
            __emit 0x4D
            __emit 0x65
            __emit 0x73
            __emit 0x73
            __emit 0x61
            __emit 0x67
            __emit 0x65
            __emit 0x42
            __emit 0x65
            __emit 0x65
            __emit 0x70
            __emit 0

    User32:
            __emit 0x75
            __emit 0x73
            __emit 0x65
            __emit 0x72
            __emit 0x33
            __emit 0x32
            __emit 0x2E
            __emit 0x64
            __emit 0x6C
            __emit 0x6C
            __emit 0
     }

    CodeEnd:;
    }

    DWORD CPEcoderDlg::PEAlign(DWORD Num,DWORD AlignTo)
    {
     /***Función para el cálculo de alineamiento ***/
     while(Num % AlignTo != 0)
     {
          Num++;
     }
     return Num;
    }
```

```
void CPEcoderDlg::WritePEFile(LPVOID pMem,HANDLE
 File,DWORD Size)
{
 /**Función para escribir los contenidos de la memoria
 en el fichero **/
 DWORD BytesWritten;

 SetFilePointer(File,0,NULL,FILE_BEGIN);
 WriteFile(File,pMem,Size,&BytesWritten,NULL);
 SetFilePointer(File,Size,NULL,FILE_BEGIN);
 SetEndOfFile(File);
}

void CPEcoderDlg::ProcessTLS(BYTE *pMem,DWORD
 VAddress,PIMAGE_NT_HEADERS pImageNT,DWORD
 FileSize,DWORD Offset)
{
 /*******Función para procesar TLS*******/
 memcpy((VOID*)(pMem+FileSize+CODE_SIZE+0x61),(pMem+Offs
  et),sizeof IMAGE_TLS_DIRECTORY32);
 pImageNT-
  >OptionalHeader.DataDirectory[9].VirtualAddress =
  VAddress+CODE_SIZE+0x61;
}

void CPEcoderDlg::ProcessPE(char* source,char* target)
{
 PIMAGE_NT_HEADERS pImageNT;
 DWORD NOBR;

 /*****Obtención de manejador del fichero fuente *****/
 HANDLE File =
  CreateFile(source,GENERIC_READ,FILE_SHARE_READ,NULL,OP
  EN_EXISTING, FILE_ATTRIBUTE_NORMAL,NULL);
 if (File == INVALID_HANDLE_VALUE)
 {
  MessageBox("¡El fichero no se puede abrir en modo
   lectura!",NULL,MB_OK | MB_ICONINFORMATION);
  return;
 }

 /*******Carga de variables globales*******/
 AddedCode();

 /*****Carga del fichero en memoria y
  reinicialización*****/
 DWORD FileSize = GetFileSize(File,NULL);
```

```
if (FileSize == 0)
{
 MessageBox("¡No están permitidos ficheros de tamaño
  nulo!",NULL,MB_OK | MB_ICONINFORMATION);
 CloseHandle(File);
 return;
}

DWORD Size = FileSize+CODE_SIZE+OTHER_SIZE+0x1000;
BYTE *pMem = new BYTE[Size];
for (DWORD i = 0; i < Size; i++)
{
 pMem[i] = 0;
}

ReadFile(File,pMem,FileSize,&NOBR,NULL);
pImageNT = ImageNtHeader((VOID*)pMem);

/*******Verificación del formato PE*******/
if (pImageNT == 0)
{
 MessageBox("¡Fichero con formato PE
  incorrecto!",NULL,MB_OK | MB_ICONINFORMATION);
 CloseHandle(File);
 delete pMem;
 return;
}

/****Verificación de la existencia de la tabla de
 importaciones ****/
DWORD ITRVA = pImageNT-
 >OptionalHeader.DataDirectory[1].VirtualAddress;
if (ITRVA == 0)
{
 MessageBox("¡Ninguna función importada?",NULL,MB_OK |
  MB_ICONINFORMATION);
 CloseHandle(File);
 delete pMem;
 return;
}

/*******Nueva sección, inicialización y redirección de
 valores necesarios *****/
DWORD NewSize = AddSection(pMem,pImageNT);
if (NewSize == 0)
{
 CloseHandle(File);
```

```
    delete pMem;
    return;
  }

  /*******Codificación de secciones en el fichero*******/
  CryptPE(pImageNT,pMem);

  /****Obtención del manejador del fichero de
   destino*****/
  HANDLE NewFile = CreateFile(target,GENERIC_WRITE |
   GENERIC_READ,FILE_SHARE_WRITE |
   FILE_SHARE_READ,NULL,CREATE_ALWAYS,FILE_ATTRIBUTE_NORM
   AL,NULL);
  if (NewFile == INVALID_HANDLE_VALUE)
  {
   MessageBox("¡No se puede crear el fichero!",NULL,MB_OK
    | MB_ICONINFORMATION);
   CloseHandle(File);
   delete pMem;
   return;
  }

  /*******Guardando el fichero*******/
  WritePEFile(pMem,NewFile,NewSize);
  MessageBox("¡Hecho!",NULL,MB_OK | MB_ICONINFORMATION);

  CloseHandle(File);
  CloseHandle(NewFile);
  delete pMem;
}

IMAGE_SECTION_HEADER
 CPEcoderDlg::RvaToSection(PIMAGE_NT_HEADERS
 pImageNT,BYTE *pMem,DWORD AdrOfSecTable,BOOL
 AdjustChar)
{
 /*****Función para obtener parámetros de la
  sección*******/
 PIMAGE_SECTION_HEADER pSection;
 IMAGE_SECTION_HEADER Section;
 DWORD vaddr;

 memcpy(&vaddr,(VOID*)(AdrOfSecTable+12),sizeof DWORD);
 pSection = ImageRvaToSection(pImageNT,pMem,vaddr);
 if (pSection == 0)
 {
```

```
    /*******Guardando parámetros de sección en variables
     *****/
    DWORD vsize,raddr,rsize,characteristics;
    memcpy(&vsize,(VOID*)(AdrOfSecTable+8),sizeof DWORD);
    memcpy(&rsize,(VOID*)(AdrOfSecTable+16),sizeof DWORD);
    memcpy(&raddr,(VOID*)(AdrOfSecTable+20),sizeof DWORD);
    memcpy(&characteristics,(VOID*)(AdrOfSecTable+36),
     sizeof DWORD);

                Section.VirtualAddress = vaddr;
                Section.Misc.VirtualSize = vsize;
                Section.PointerToRawData = raddr;
                Section.SizeOfRawData = rsize;

                if (AdjustChar)
                Section.Characteristics = characteristics |
                 0x80000000;

                return Section;
    }
    if (AdjustChar)
    pSection->Characteristics = pSection->Characteristics |
     0x80000000;

    Section = *pSection;
    return Section;
}
void  CPEcoderDlg::SimpleCrypt(BYTE *pInput,DWORD lSize)
{
    /*******Algoritmo sencillo de codificación*******/
    for (DWORD lCount = 0; lCount < lSize ;
     lCount++,pInput++)
     {
      *pInput ^= 0xAB;
     }
}
void CPEcoderDlg::AssembleIT(BYTE *pMem,DWORD
 NewSectionRAW,DWORD NewSectionVA,PIMAGE_NT_HEADERS
 pImageNT)
{
    /***Función para crear una tabla de importaciones
     nueva***/
    DWORD *Name = new DWORD;
    DWORD *FirstThunk = new DWORD;
    DWORD *ThunkLoadLibrary = new DWORD;
    DWORD *ThunkGetProcAddress = new DWORD;
```

```
NewSectionRAW += CODE_SIZE;
NewSectionVA += CODE_SIZE;

/**Creación de estructuras de la tabla de
 importaciones**/
*Name = NewSectionVA+2*sizeof
 IMAGE_IMPORT_DESCRIPTOR+3*sizeof
 IMAGE_THUNK_DATA+12+1+14+1+2*sizeof WORD;
*FirstThunk = NewSectionVA+2*sizeof
 IMAGE_IMPORT_DESCRIPTOR;

memcpy((VOID*)(pMem+NewSectionRAW+12),Name,sizeof
 DWORD);
memcpy((VOID*)(pMem+NewSectionRAW+16),FirstThunk,
 sizeof DWORD);

*ThunkLoadLibrary = NewSectionVA+2*sizeof
 IMAGE_IMPORT_DESCRIPTOR+3*sizeof IMAGE_THUNK_DATA;
*ThunkGetProcAddress = NewSectionVA+2*sizeof
 IMAGE_IMPORT_DESCRIPTOR+3*sizeof
 IMAGE_THUNK_DATA+sizeof WORD+12+1;

memcpy((VOID*)(pMem+NewSectionRAW+2*sizeof
 IMAGE_IMPORT_DESCRIPTOR),(VOID*)ThunkLoadLibrary,sizeo
 f IMAGE_THUNK_DATA);
memcpy((VOID*)(pMem+NewSectionRAW+2*sizeof
 IMAGE_IMPORT_DESCRIPTOR+sizeof
 IMAGE_THUNK_DATA),(VOID*)ThunkGetProcAddress,sizeof
 IMAGE_THUNK_DATA);

/*******LoadLibraryA*******/
DWORD Adresa = NewSectionRAW+2*sizeof
 IMAGE_IMPORT_DESCRIPTOR+3*sizeof
 IMAGE_THUNK_DATA+sizeof WORD;
pMem[Adresa] = 0x4C;
pMem[Adresa+1] = 0x6F;
pMem[Adresa+2] = 0x61;
pMem[Adresa+3] = 0x64;
pMem[Adresa+4] = 0x4C;
pMem[Adresa+5] = 0x69;
pMem[Adresa+6] = 0x62;
pMem[Adresa+7] = 0x72;
pMem[Adresa+8] = 0x61;
pMem[Adresa+9] = 0x72;
pMem[Adresa+10] = 0x79;
pMem[Adresa+11] = 0x41;
```

```
 /*******GetProcAddress*******/
 Adresa = NewSectionRAW+2*sizeof
  IMAGE_IMPORT_DESCRIPTOR+3*sizeof
  IMAGE_THUNK_DATA+2*sizeof WORD+12+1;
 pMem[Adresa] = 0x47;
 pMem[Adresa+1] = 0x65;
 pMem[Adresa+2] = 0x74;
 pMem[Adresa+3] = 0x50;
 pMem[Adresa+4] = 0x72;
 pMem[Adresa+5] = 0x6F;
 pMem[Adresa+6] = 0x63;
 pMem[Adresa+7] = 0x41;
 pMem[Adresa+8] = 0x64;
 pMem[Adresa+9] = 0x64;
 pMem[Adresa+10] = 0x72;
 pMem[Adresa+11] = 0x65;
 pMem[Adresa+12] = 0x73;
 pMem[Adresa+13] = 0x73;

 /*******kernel32.dll*******/
 Adresa = NewSectionRAW+2*sizeof
  IMAGE_IMPORT_DESCRIPTOR+3*sizeof
  IMAGE_THUNK_DATA+2*sizeof WORD+12+1+14+1;
 pMem[Adresa] = 0x6B;
 pMem[Adresa+1] = 0x65;
 pMem[Adresa+2] = 0x72;
 pMem[Adresa+3] = 0x6E;
 pMem[Adresa+4] = 0x65;
 pMem[Adresa+5] = 0x6C;
 pMem[Adresa+6] = 0x33;
 pMem[Adresa+7] = 0x32;
 pMem[Adresa+8] = 0x2E;
 pMem[Adresa+9] = 0x64;
 pMem[Adresa+10] = 0x6C;
 pMem[Adresa+11] = 0x6C;

 delete ThunkGetProcAddress;
 delete ThunkLoadLibrary;
 delete FirstThunk;
 delete Name;
}

void CPEcoderDlg::CryptPE(PIMAGE_NT_HEADERS
 pImageNT,BYTE *pMem)
{
```

```
/****Función para codificar secciones en el
 fichero****/
IMAGE_SECTION_HEADER *pSection = new
 IMAGE_SECTION_HEADER;
DWORD SecNum = pImageNT->FileHeader.NumberOfSections;
DWORD AdrOfSecTable;

for (DWORD i = 0; i < SecNum; i++)
{
 AdrOfSecTable = (DWORD)pImageNT+(sizeof
  IMAGE_NT_HEADERS)+i*(sizeof IMAGE_SECTION_HEADER);
 *pSection =
  RvaToSection(pImageNT,pMem,AdrOfSecTable,FALSE);
 if (*(LPDWORD)pSection->Name != 0x77656E2E /*".new"*/
  &&
 *(LPDWORD)pSection->Name != 0x7273722E /*".rsr"*/ &&
 *(LPDWORD)pSection->Name != 0x63727372 /*"rsrc"*/ &&
 *(LPDWORD)pSection->Name != 0x6C65722E /*".rel"*/ &&
 *(LPDWORD)pSection->Name != 0x6F6C6572 /*"relo"*/ &&
 *(LPDWORD)pSection->Name != 0x6164652E /*".eda"*/ &&
 *(LPDWORD)pSection->Name != 0x736C742E /*.tls*/ &&
 pSection->PointerToRawData != 0 && pSection-
  >SizeOfRawData != 0 )
 {
 DWORD Size = (pSection->Misc.VirtualSize < pSection-
  >SizeOfRawData ? pSection->Misc.VirtualSize :
  pSection->SizeOfRawData);
 SimpleCrypt(pMem+pSection->PointerToRawData,Size);
 }
}
delete pSection;
}


DWORD CPEcoderDlg::AddSection(BYTE
 *pMem,PIMAGE_NT_HEADERS pImageNT)
{
 /*******Función para añadir, inicializar una sección
  nueva y redirigir valores concretos*******/
 IMAGE_SECTION_HEADER *pSection = new
  IMAGE_SECTION_HEADER;
 DWORD SecNum = pImageNT->FileHeader.NumberOfSections;
 DWORD SecSize = sizeof IMAGE_SECTION_HEADER;
 DWORD Size = (SecNum+1)*SecSize;
 DWORD TotalSize = (DWORD)pImageNT-
  (DWORD)pMem+Size+sizeof IMAGE_NT_HEADERS;
```

```
DWORD HeadersSize = pImageNT-
 >OptionalHeader.SizeOfHeaders;

/****Comprobación de espacio en la tabla de
 secciones****/
if (TotalSize > HeadersSize)
{
 MessageBox(";Espacio insuficiente en la tabla de
  secciones!",NULL,MB_OK | MB_ICONINFORMATION);
 return 0;
}
/****Obtención de los parámetros de la última
 sección****/
DWORD AdrOfSecTable;
for (DWORD i = 0; i < SecNum; i++)
{
 AdrOfSecTable = (DWORD)pImageNT+sizeof
  IMAGE_NT_HEADERS+i*(sizeof IMAGE_SECTION_HEADER);
 *pSection =
  RvaToSection(pImageNT,pMem,AdrOfSecTable,TRUE);
}

unsigned char Name1[8] = ".new";
DWORD *VirtualAddress = new DWORD;
DWORD *VirtualSize = new DWORD;
DWORD *SizeOfRawData = new DWORD;
DWORD *PointerToRawData = new DWORD;
DWORD *Characteristics = new DWORD;

/*****Cálculo de los parámetros de la sección
 nueva*****/
*VirtualAddress = PEAlign(pSection-
 >VirtualAddress+pSection->Misc.VirtualSize,pImageNT->
 OptionalHeader.SectionAlignment);
*VirtualSize = CODE_SIZE+OTHER_SIZE;
*SizeOfRawData = PEAlign(CODE_SIZE+OTHER_SIZE,pImageNT-
 >OptionalHeader.FileAlignment);
*Characteristics = 0xE00000E0;
*PointerToRawData = pSection-
 >PointerToRawData+pSection->SizeOfRawData;

/*******Inicializando el campo de las variables*******/
VAR[0] = *VirtualAddress;
VAR[1] = pImageNT->OptionalHeader.AddressOfEntryPoint;
VAR[2] = pImageNT-
 >OptionalHeader.DataDirectory[1].VirtualAddress;
```

```
DWORD RealSize = *SizeOfRawData+*PointerToRawData;
DWORD Adresa = AdrOfSecTable+sizeof
IMAGE_SECTION_HEADER;

/*******Grabando la definición de la sección nueva en
 la tabla de secciones*******/
memcpy((VOID*)(Adresa),Name1,sizeof Name1);
memcpy((VOID*)(Adresa+8),(VOID*)VirtualSize,sizeof
 DWORD);
memcpy((VOID*)(Adresa+12),(VOID*)VirtualAddress,sizeof
 DWORD);
memcpy((VOID*)(Adresa+16),(VOID*)SizeOfRawData,sizeof
 DWORD);
memcpy((VOID*)(Adresa+20),(VOID*)PointerToRawData,sizeo
 f DWORD);
memcpy((VOID*)(Adresa+36),(VOID*)Characteristics,sizeof
 DWORD);

/*******Insertando el código de control*******/
memcpy((VOID*)(pMem+*PointerToRawData),(VOID*)
 (CODE_START),CODE_SIZE);

/*******Insertando variables******/
memcpy((VOID*)(pMem+*PointerToRawData+VAR_START),(VOID*
 )VAR,sizeof VAR);

/*******Corrigiendo valores***********/
pImageNT->FileHeader.NumberOfSections += 1;
pImageNT->OptionalHeader.SizeOfImage += *SizeOfRawData;
pImageNT->OptionalHeader.SizeOfCode += *SizeOfRawData;
pImageNT->OptionalHeader.SizeOfInitializedData +=
 *SizeOfRawData;

/*******Procesando la tabla de importaciones *******/
AssembleIT(pMem,*PointerToRawData,*VirtualAddress,
 pImageNT);
pImageNT-
 >OptionalHeader.DataDirectory[1].VirtualAddress =
 *VirtualAddress+CODE_SIZE;

/*******TLS*******/
if(pImageNT-
 >OptionalHeader.DataDirectory[9].VirtualAddress != 0)
{
 PIMAGE_SECTION_HEADER Section =
  ImageRvaToSection(pImageNT,pMem,pImageNT->
  OptionalHeader.DataDirectory[9].VirtualAddress);
```

```
    DWORD Offset = pImageNT-
      >OptionalHeader.DataDirectory[9].VirtualAddress-
      Section->VirtualAddress+Section-> PointerToRawData;

    ProcessTLS(pMem,*VirtualAddress,pImageNT,*PointerToRaw
      Data,Offset);
  }

  /*******Redirigiendo Program Entry Point*******/
  pImageNT->OptionalHeader.AddressOfEntryPoint =
    *VirtualAddress;

  delete Characteristics;
  delete PointerToRawData;
  delete SizeOfRawData;
  delete VirtualSize;
  delete VirtualAddress;
  delete pSection;

  return RealSize;
}

void CPEcoderDlg::OnOk()
{
  /*******Selección del fichero fuente y destino *******/
  OPENFILENAME Open;
  char Source[512] = "";

  Open.lStructSize = 76;
  Open.hwndOwner = NULL;
  Open.hInstance = NULL;
  Open.lpstrFilter = "EXE files (*.EXE)\0*.EXE\0\0";
  Open.lpstrCustomFilter = NULL;
  Open.nMaxCustFilter = NULL;
  Open.nFilterIndex = NULL;
  Open.lpstrFile = Source;
  Open.nMaxFile = 512;
  Open.lpstrFileTitle = NULL;
  Open.nMaxFileTitle = NULL;
  Open.lpstrInitialDir = NULL;
  Open.lpstrTitle = "Por favor, indique el fichero que
    desea codificar";
  Open.Flags = OFN_FILEMUSTEXIST | OFN_HIDEREADONLY |
    OFN_EXPLORER | OFN_PATHMUSTEXIST;
  Open.nFileOffset = NULL;
  Open.nFileExtension = NULL;
  Open.lpstrDefExt = NULL;
```

```
    Open.lCustData = NULL;
    Open.lpfnHook = NULL;
    Open.lpTemplateName = NULL;

    BOOL Selected = GetOpenFileName(&Open);

    if (Selected)
    {
     OPENFILENAME Open2;
     char Target[512] = "";

     Open2.lStructSize = 76;
     Open2.hwndOwner = NULL;
     Open2.hInstance = NULL;
     Open2.lpstrFilter = "All files (*.*)\0*.*\0\0";
     Open2.lpstrCustomFilter = NULL;
     Open2.nMaxCustFilter = NULL;
     Open2.nFilterIndex = NULL;
     Open2.lpstrFile = Target;
     Open2.nMaxFile = 512;
     Open2.lpstrFileTitle = NULL;
     Open2.nMaxFileTitle = NULL;
     Open2.lpstrInitialDir = NULL;
     Open2.lpstrTitle = "Guardar a...";
     Open2.Flags = OFN_OVERWRITEPROMPT | OFN_HIDEREADONLY |
      OFN_EXPLORER;
     Open2.nFileOffset = NULL;
     Open2.nFileExtension = NULL;
     Open2.lpstrDefExt = NULL;
     Open2.lCustData = NULL;
     Open2.lpfnHook = NULL;
     Open2.lpTemplateName = NULL;

     BOOL Selected2 = GetOpenFileName(&Open2);

     if(Selected2)
     {
      ProcessPE(Source,Target);
     }
    }
}
```

PROTECCIONES ALTERNATIVAS

Cargador de símbolos AntiSoftICE

Aunque éste sea un viejo y bien conocido truco, figura aquí para completar la lista de posibilidades en este campo. Pretende evitar que SoftICE interrumpa el programa en la dirección del Program Entry Point una vez que lo arranque el cargador de símbolos.

Resulta muy sencillo de utilizar. El cargador de símbolos interrumpirá el programa en la dirección del `Program Entry Point` sólo si esta dirección fuera sustituida en una sección cuyas características contuviese el indicador `contains excecutable code`. Por lo tanto, si se alterasen las características de esta sección (sección con código ejecutable, cuyo nombre normalmente es .next, CODE, etc.) con algún editor PE (o directamente) para comprobar que no contiene el indicador anteriormente mencionado, el problema quedaría resuelto. Posiblemente la forma más cómoda de llevar esto a cabo sea mediante un editor PE denominado PEditor, incluido en el CD adjunto.

Figura 7-11. Comodidad para definir las características de las secciones con PEditor

Dada su fama, no deben ponerse grandes esperanzas en este método. Ahora bien, si se incluye la comprobación de las características de una sección, puede ofrecer buenos resultados.

Comprobación del punto de entrada al programa

Ya se ha mencionado con anterioridad la gran importancia que tiene el valor del `Program Entry Point` original. Lo que faltaba por indicar es que el valor del `Program Entry Point` actual puede emplearse, sencillamente, para saber si el fichero se ha descodificado.

Su lógica es muy sencilla. Cuando el programa se descodifique, el `Program Entry Point` volverá a su valor original antes descodificarse. Bastará con situar la comprobación del valor actual y del nuevo `Program Entry Point` creado por el codificador PE en algún punto del programa. Si estos dos valores no fueran iguales, sería obvio que el programa sí se ha descodificado y que el `Program Entry Point` ha cambiado a su valor original. En este caso, la mejor solución pasaría por provocar un error en el programa para hacer creer al posible cracker que ha cometido un error en la descodificación.

Nunca deberá compararse el valor correcto de `Program Entry Point` con el actual. De esa manera se podría hallar el valor original directamente a partir del programa. Debe compararse el valor actual de `Program Entry Point` con el nuevo valor creado por el codificador PE.

Ahora bien, esta comprobación se puede sortear con facilidad situando una instrucción de bifurcación al `Program Entry Point` original en la dirección del `Program Entry Point` creada por el codificador PE. No obstante, al ser tan infrecuente este tipo de comprobación en los programas, resultará muy improbable que la contemple el posible cracker.

RSA

Este algoritmo de codificación asimétrica, creado entre los años 1977 y 1978 por Ron Rivest, Adi Shamir y Leonard Adleman, constituye uno de los estándares no oficiales de hoy en día. La seguridad de este algoritmo radica en la gran dificultad para obtener grandes números a partir de la multiplicación de otros números primos grandes, la solución dependerá, por lo tanto, del cálculo de los factores del producto. Su seguridad es proporcional a la longitud de la clave. Cualquier grupo o empresa especializada podrá anular fácilmente claves con una longitud de 384 bits. Aunque resulte difícil comprobarlo, se considera que una clave con una longitud de 1024 bits es lo suficientemente buena como para defenderse incluso de las entidades gubernamentales.

Actualmente existen muchos algoritmos rápidos y seguros, como los criptosistemas elípticos (ECC), que obtienen un nivel de seguridad comparable a RSA con una clave de longitud de 2048 bits aunque utiliza una clave de 160 a 180 bits de longitud. Todo ello

indica que la criptografía está constantemente evolucionando y que los antiguos algoritmos, como el RSA, van perdiendo su utilidad.

No se detallará aquí el concepto matemático específico de RSA ni el funcionamiento real de este algoritmo. Sí se indicará, sin embargo, el procedimiento para crear una pareja de claves correcta en un par de pasos con ejemplos de algoritmos útiles (aptos incluso para aquellos que no sean muy aficionados a las matemáticas) con ejemplos prácticos para codificar y descodificar datos mediante RSA.

1. En primer lugar, definanse dos números primos diferentes, P y Q. Estos números se emplearán posteriormente para calcular los valores de las claves. El múltiplo de P y Q, con frecuencia denominado N, será la parte pública y también la parte privada de la clave. Ello explica la importancia de seleccionar números primos tan grandes como sea posible para procurar que sea muy difícil hallar los números primos P y Q a partir de N y poder así anular el código. Con objeto de evitar un procedimiento excesivamente largo, no se emplearán números primos de, por ejemplo, 1024 bits, sino números y claves menores.

A continuación se muestra una función que obtiene el número primo mayor más próximo a otro dado:

```
__int64 CRSADlg::GetPrime(__int64 start)
{
__int64 i,Num,Sqrt;

if (start <= 0 || start == 1 || start == 2 )
            return 2;
else
{
      Num = start;
      if (Num % 2 == 0)    //  ¿número par?
            Num++;    //  en caso afirmativo, sustituir
                      // por impar

      Sqrt = (__int64)sqrt (Num);    //  raíz
      for (i = 3; i <= Sqrt; i += 2)    //  i =i+2 -
                                 // sólo números
                                   // impares
      {
            if (Num % i == 0)    //  ¿número divisible
                                 // sin resto?
            {
                  i =1;
                  Num += 2;
                  Sqrt = (__int64)sqrt (Num);
            }
```

```
            else if (i == Sqrt)    //  ¿hallado número
                                   // primo?
                break;
        }
        return Num;
    }
}
```

2. En segundo lugar, determínese el valor de la parte pública de la clave (para codificar) bajo E; este valor deberá ser menor al múltiplo de (P-1)(Q-1) y el mayor común divisor a estos dos números deberá ser 1. Lógicamente significa que debe ser un número impar.

La función siguiente obtendrá el mayor divisor común de dos números dados:

```
__int64 CRSADlg::GetLCD(__int64 a,__int64 b)
{

/***Posibilidad alternativa de procesar números
 negativos***/
//  a = abs (a);
//  b = abs (b);

/******* las siguientes dos opciones serán imposibles si
 los números a y b son primos*******/
 if (a == 0 || b == 0)
      return 0;

 if (a == 1 || b == 1)
      return 1;

 while (a != b)   //  cálculo del mayor divisor común
 {
      if (a < b)
           b = b-a;

      if (a > b)
           a = a-b;
 }
 return a;
}
```

Ya resulta factible crear un bucle que calcule el número E correcto superior y más próximo a otro dado (en el ejemplo, indicado por `determined_number`), siendo 1 el mayor común divisor de los números E y (P-1)(Q-1).

```
dwhile (determined_number < (p-1)*(q-1))
{
 if (GetLCD (determined_number,(p-1)*(q-1)) == 1)
 {
       //  hallado número E
       //  E = determined_number;
       break;
 }
 determined_number++;
}
```

3. Calcúlese el valor de la parte privada de la clave D (para descodificar); DE-1 deberá ser divisible sin resto por (P-1)(Q-1). La expresión matemática de esta operación será la siguiente:

```
DE = 1 (mod (P-1)(Q-1))
```

Debe encontrarse un número X entero para quien D sea uno número entero y cumpla la expresión siguiente:

D = (X(P-1)(Q-1) + 1)/E

La función siguiente obtendrá el número D y utilizará los valores iniciales y finales para calcular el número X, utilizando P, Q y E como parámetros:

```
__int64 CRSADlg::GetD(__int64 start,__int64 end,__int64
 p,__int64 q,__int64 e)
{
 __int64 x,k = (p-1)*(q-1);

/*******En caso de valores negativos para las variables
 iniciales y finales *******/
//  start = abs (start);
//  end = abs (end);

 if (start >= end)
 {
      MessageBox(" El valor inicial para determinar el
       múltiplo del número X deberá ser inferior al
       valor final",NULL,MB_OK |MB_ICONINFORMATION);
      return 0;
 }
 if (start == 0)
      start = 1;
```

```
for (x = start; x <= end; x++)
{
      if ((x*k+1) % e == 0)    //  ¿hallado número D?
      {
            __int64 d = (x*k+1) / e;
            return d;
      }
}
return 0;
}
```

4. Supóngase que T representa los datos que se han de codificar. Estos datos se codificarán de la manera siguiente: (T^E) mod PQ.

5. Supóngase que C representa los datos codificados que se han de descodificar. Estos datos se descodificarán de la manera siguiente: (C^D) mod PQ.

Bastará con emplear una sencilla calculadora basada en los algoritmos anteriormente mencionados para hallar la pareja de claves necesaria en RSA. Si aún no se supiera cómo o se prefiriera no hacerlo, este programa se podrá encontrar además de su código fuente en el CD adjunto.

Ejemplo de aplicación con RSA

P = 61

Q = 53

PQ = 3233

E = 17

D = 2753

Parte pública de la clave: (PQ,E) = (3233,17)

Parte privada de la clave: (PQ,D) = (3233,2753)

El número 123 se codificará de la siguiente manera:

```
encrypt(123) = (123 ^17) mod 3233
      =337587917446653715596592958817679803 mod 3233
      =855
```

La descodificación será:

```
decrypt(855)=(855 ^2753)mod 3233 =
504328889584160687344228991273944663145387836003550 93
 155549675645010556286120825599787442454281100543834 98
 654289336384930246451441507851720917966547826353070 99
 638035387326500896686074771829745822950342950407903 58
 184594095637793858659893688380836028401325097686207 66
 977396675332505428260934757351379880632564826393344 53
 092594385562429233017519771900169249169128091505960 19
 178760171349725439279215696701789902134307146468971 27
 961027718137839458696772898693423652403116932170892 69
 617643726521315665833158 7...(el número entero
 necesitaría un par de páginas)mod 3233 =123
```

CONCLUSIÓN SOBRE EL FORMATO PE Y COMPRESORES–CODIFICADORES PE

Así se llega al final del capítulo más largo de este libro. Supone para el lector una información básica sobre el formato PE que le permitirá ahondar e investigar este tema en mayor profundidad. Si así lo hiciese, descubrirá un gran número de aspectos interesantes y otras posibilidades del formato PE que no se han mencionado aquí.

Ahora el lector ya podrá crear su propio codificador PE y añadirle varias protecciones descritas en otros capítulos.

Los compresores y codificadores PE figuran entre los métodos más utilizados de protección y constituyen la base de la inmensa mayoría de las protecciones comerciales. Mucha gente comete el error de considerar su programación muy compleja y prefieren emplear productos específicos aunque sean conscientes de sus limitaciones.

Gracias a este libro, el primero en su género, ya no resulta necesario, puesto que aporta una guía completa para poder programar un codificador PE con sus correspondientes aclaraciones y deshacer el misterio y la información errónea que injustamente se han asociado a este tema.

No debe creerse ciegamente a nadie que afirme que su protección basada en un codificador PE sea algo increíblemente complicado y muy difícil de crear por lo que merezca la pena pagar desproporcionadas cantidades de dinero. Habiendo considerado detenidamente todas las posibilidades anteriormente mencionadas de protección, el lector podrá crear su propio codificador PE de mejor calidad que la mayoría de los que ya se encuentran en el mercado. Por fin, las empresas "profesionales" especializadas

en el software de protección tendrán algún tipo de competencia y se verán en la necesidad de considerar seriamente producir y vender productos genuinos de alta calidad en vez de sistemas de seguridad elementales vendidos como productos de los mejores desarrolladores especialistas en sistemas antipirateo, como ha sido el caso hasta ahora, debido a la falta de conocimiento público. Sería deseable que esta situación cambiara y que ya nadie más pudiese ganar cifras astronómicas de dinero gracias a la ignorancia de la gente.

La verdad es que los desarrolladores siempre se encuentran un paso por detrás de los crackers, nunca llegan a alcanzarlos; por eso siempre es mejor desarrollarlo todo uno mismo. A diferencia de los desarrolladores, el lector aprenderá entonces de sus errores (o ésa es la esperanza del autor) y no los volverá a repetir.

CAPÍTULO 8

OTROS PROGRAMAS UTILIZADOS POR LOS CRACKERS

REGISTRY MONITOR

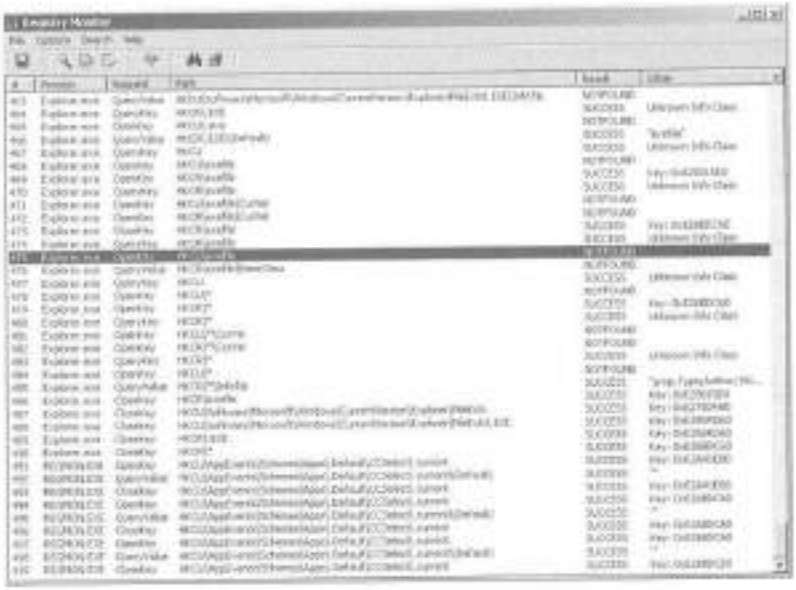

Figura 8-1. Seguimiento realizado por Registry Monitor

El uso principal de este programa estriba en la capacidad que tiene para supervisar todos los accesos efectuados al registro; se suele utilizar para rastrear los mecanismos de protección que bien lean o guarden información en el registro. Entre ellos figuran, por ejemplo, los programas de duración limitada o con otro tipo de restricción que almacenan en el registro o cuándo se instalaron o cuántas veces cierta función se lleva utilizando.

Algunos desarrolladores, al intentar ocultar (a los crackers) las definiciones que un programa realiza en el registro de Windows, utilizan áreas que normalmente no tienen ese cometido, como el espacio asignado al sistema, otros programas ya instalados, etc. El hecho de que Registry Monitor detecte con facilidad estas operaciones demuestra de nuevo la ingenuidad de algunos programadores.

A continuación se mostrará el manejo de este programa y alguna de sus funciones mediante un sencillo ejemplo extraído del capítulo 2 sobre depuración donde se rastrean algunos valores del registro para detectar SoftICE:

```
HKEY Key;
BYTE VerDataBuffer[200],InstDataBuffer[200];
DWORD VerSize = 5,InstSize = 128;

if
 (RegOpenKeyEx(HKEY_LOCAL_MACHINE,"Software\\NuMega\\SoftI
 CE\\",NULL,KEY_READ,&Key) == ERROR_SUCCESS)
                    // ¿hallada clave añadida por SoftICE?
{
 RegQueryValueEx(Key,"Current
  Version",NULL,NULL,VerDataBuffer,&VerSize);
                                        // Versión de SoftICE
 RegQueryValueEx(Key,"InstallDir",NULL,NULL,
  InstDataBuffer,&InstSize);
                   // Directorio de instalación de SoftICE
 MessageBox((const char*)InstDataBuffer,(const
  char*)VerDataBuffer,MB_OK);
}
else

    MessageBox("SoftICE no detectado",NULL,MB_OK);
```

Arránquese Registry Monitor y a continuación el programa de detección de SoftICE. No se pulse todavía el botón para comenzar la detección. Obsérvese lo que Registry Monitor ya ha detectado.

Un buen número de procesos, como el Explorador, interactúan casi constantemente con el registro dificultando cualquier búsqueda en el Registry Monitor. Afortunadamente

esta herramienta contiene un filtro que ahorra al usuario tener que buscar entre cientos, sino miles, de definiciones realizadas en el registro para encontrar las que esté buscando. Púlsese Ctrl+E para desactivar la supervisión, la herramienta dejará entonces de mostrar los accesos al registro realizados por los distintos procesos en ejecución. Sabidos los nombres de los procesos que acceden al registro, podrán definirse a la herramienta aquellos que se desean observar; e incluso si no se supieran tales nombres, queda la alternativa, bastante más larga, de excluir los demás procesos. Realizada la labor de búsqueda, se podrá observar el nombre del proceso bajo la columna "Process". El proceso que detecta SoftICE se denomina RegisterSICheck. Púlsese ahora Ctrl+L (o bien selecciónese Options Filter...) para mostrar el cuadro de diálogo del filtrado.

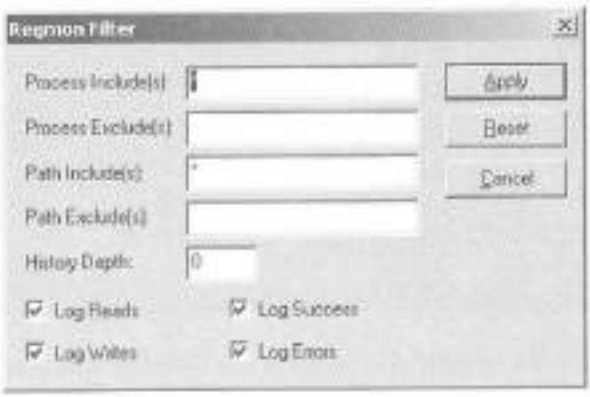

Figura 8-2. Cuadro de diálogo para realizar el filtrado

En la anterior figura se muestra el cuadro de diálogo donde se definen los distintos criterios para realizar el filtrado de los procesos mostrados. Introdúzcase el nombre del proceso (RegisterSiCheck) en el campo "Include" y púlsese por último "Apply".

Vuélvase a activar la supervisión de Registry Monitor, suprímanse los procesos anteriores pulsando Ctrl+X y ejecútese el programa para detectar SoftICE de nuevo. Al incluir solamente los accesos al registro de las aplicaciones seleccionadas, la lista resulta mucho más corta y clara. A continuación púlsese el botón para arrancar la detección de SoftICE, Registry Monitor visualizará inmediatamente las líneas siguientes:

```
RegistrSICheck.  OpenKey HKLM \Software \NuMega \SoftIce \
----------------------------------------------------------
RegistrSICheck.  QueryValue      HKLM \Software \NuMega \SoftIce \Current Version
----------------------------------------------------------
RegistrSICheck.  QueryValue      HKLM \Software \NuMega \SoftIce \Current Version
----------------------------------------------------------
RegistrSICheck.  QueryValue      HKLM \Software \NuMega \SoftIce \InstallDir
----------------------------------------------------------
RegistrSICheck.  QueryValue      HKLM \Software \NuMega \SoftIce \InstallDir
----------------------------------------------------------
```

Queda así al descubierto el intento de detectar SoftICE a través del registro.

FILE MONITOR

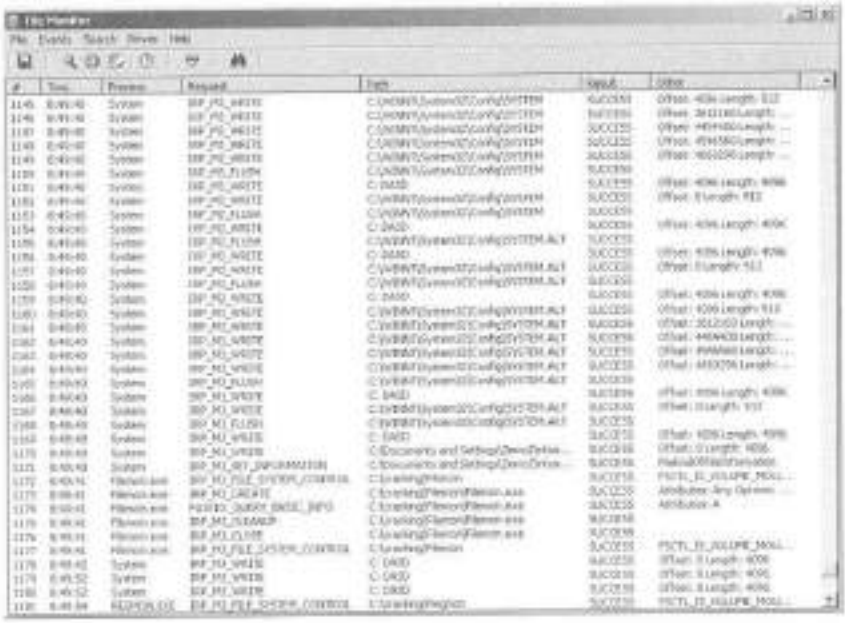

Figura 8-3. La interfaz de usuario de File Monitor y de Registry Monitor resultan prácticamente idénticas

Este otro programa también supervisa ciertas actividades de los procesos en ejecución. En este caso, los accesos a ficheros. File Monitor, al igual que el anterior programa, resulta útil para anular muchos tipos de protecciones. Puesto que, dejando al margen que se supervisa un tipo de actividad diferente, apenas existen diferencias entre este programa y Registry Monitor, sólo se incluirá aquí un ejemplo sencillo de su manejo; en este caso se buscará el nombre correcto de un fichero de claves de un programa protegido con este método. El código del programa es verdaderamente sencillo, tan sólo es una fracción de la posible protección que compruebe la presencia de un fichero con un nombre concreto. Como botón de muestra, puede resultar suficiente.

```
HANDLE File = CreateFile("key_file.key ",
 GENERIC_READ,FILE_SHARE_READ,NULL,OPEN_EXISTING,
 FILE_ATTRIBUTE_NORMAL,NULL);  // obtención del manejador
                                                  // del fichero
if (File == INVALID_HANDLE_VALUE)
{
 MessageBox("Error: no hallado fichero de claves",
  NULL,MB_OK);
 return;
}
```

Como con el ejemplo del Registry Monitor, apliquese un filtro al proceso correspondiente para facilitar la búsqueda. Supongamos que el proceso en cuestión se denomina FileChecker.exe. File Monitor mostrará lo siguiente nada más intentar comprobar la presencia del fichero:

```
FileCheck.exe    IRP_MJ_FILE_SYSTEM_CONTROL      path                    FileCheck.exe
-------------------------------------------------------------------------------------
        IRP_MJ_CREATE                          path \key_file.key
-------------------------------------------------------------------------------------
```

Tan sólo resta leer el nombre del fichero que se está buscando.

"Result", la segunda columna empezando por la derecha, mostrará el resultado de la operación realizada. Si el fichero buscado no llegara a localizarse, la columna mostraría el mensaje "FILE NOT FOUND", en caso contrario: "SUCCESS".

R!SC'S PROCESS PATCHER

Este programa genera cargadores rápidos, por ello será especialmente apreciado por quienes no les guste programar su propio cargador o bien carezcan de experiencia para realizarlo. Funciona mediante varios ficheros de mandatos que se emplean para generar el cargador.

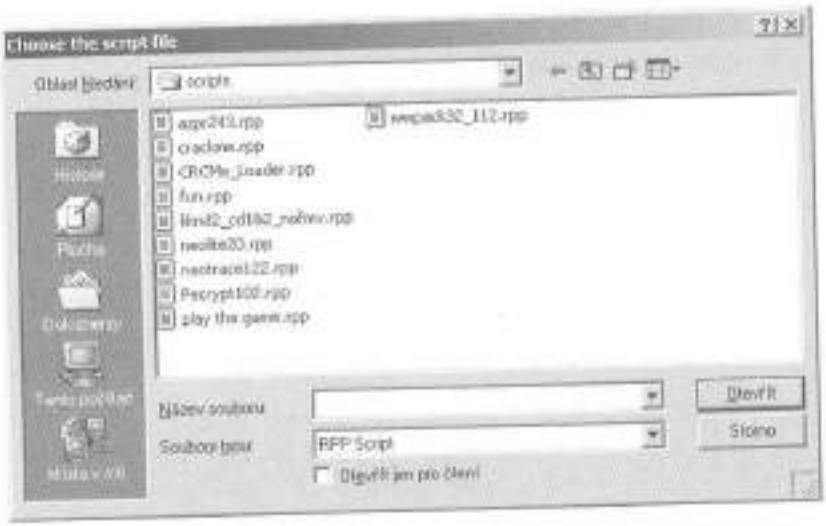

Figura 8-4. No existen ni cuadros de diálogo ni menús en este programa

Basta tan sólo con arrancar el programa y elegir el fichero de mandatos. Véase la lista de mandatos y los dos ejemplos siguientes.

Ficheros de mandatos

- ; – indica comentario. Queda ignorado todo lo que siga hasta la línea siguiente.
- F – nombre del fichero o proceso que ejecutará o modificará el cargador.
- O – nombre del fichero cargador.
- P – define parámetros necesarios para realizar modificaciones de partes de la memoria donde se carga el programa. Se aplica la sintaxis siguiente: ADDRESS/SEARCH/REPLACE, que significa: dirección de memoria donde se realizarán las modificaciones + carácter "/" + bytes en esta dirección (para verificar la validez, útil, por ejemplo, al comprobar si el fichero se ha descodificado) separados por comas + otro carácter "/" + bytes con las modificaciones.
- T – número de intentos en modificar la memoria de un proceso. Resulta útil, por ejemplo, al alterar programas protegidos por compresores o codificadores PE, donde no se pueden practicar las modificaciones antes de que el fichero sea descodificado o descomprimido.
- R – significa 'proseguir hilo' (en inglés, "Resume Thread"), esto es, rearrancar el proceso con el estado "suspended".
- : – carácter empleado al final de cada mandato.
- $ – indica el final de un fichero de mandatos.

He aquí un ejemplo de fichero de mandatos:

```
;CiA 's Trial CrackMe 0.99b
;splash screen crack by R!SC april 1999

F=crackme.exe:           ; nombre del fichero que va a
                        ;ejecutarse y modificarse
O=crack_the_crackme.exe:; nombre del fichero de
                        ; carga
P=00465F18/83/C3:  ; modificación del fichero en memoria
                  ; que efectuará el cargador
                   ; incluirá la instrucción RET al
                 ; principio de una rutina  que
               ; muestre la imagen de bienvenida
$
```

```
;R!SC 's cdcheck crackme crack,by R!SC - june 1999

F=play the game.exe:     ; nombre del fichero que va a
                       ; ejecutarse y modificarse

O=crack checkcd.exe:     ; nombre del fichero de carga

P=004014B2/74,02/72,00: ;anti-SoftICE
P=00401202/85,C0/B0,01: ;¿se ha insertado un disco?
P=0040121C/75,5D/75,00: ; verificación del nombre del
                       ; disco
P=00401517/75,43/75,00: ; comprobación de integridad
                                          ;(checksum)
$
```

Una vez definido el fichero sólo queda ejecutar el programa y elegir el fichero de mandatos, ya se habrá creado el cargador.

THE CUSTOMISER

Obsérvese la siguiente imagen de la calculadora de Windows:

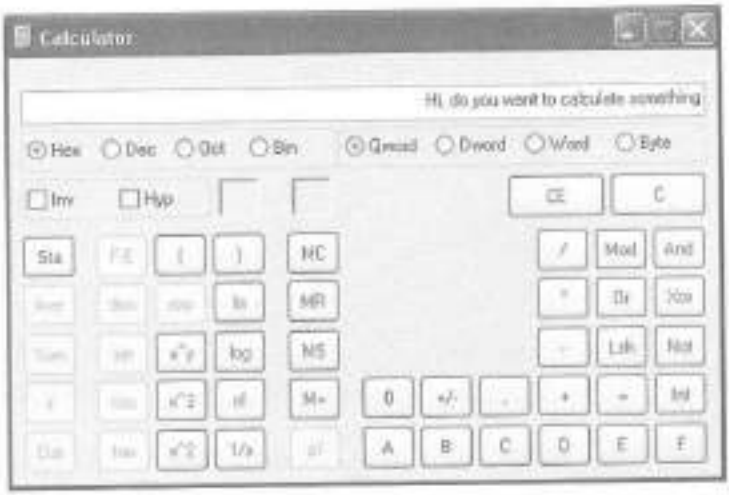

Figura 8-5. Imposible efectuar operaciones con esta calculadora

Bien puede apreciarse que esta calculadora no es como las demás. Ha sido modificada por uno de los muchos editores de recursos. Aquí se describirá uno de ellos. Denominado "The Customiser", es probablemente el editor de recursos en tiempo de ejecución más conocido.

Este programa permite trabajar y manipular todas las ventanas que estén activas en un momento dado en Windows. No importa que sean cuadros de diálogo, ventanas, botones, campos para marcar opciones o cualquier otro control. Todo se puede editar; este programa permite hacer al usuario lo que quiera. Se pueden modificar los textos de los controles y ventanas, activarlos o desactivarlos, desplazarlos, mostrarlos o ocultarlos, y muchas más cosas. Las modificaciones podrán entonces guardarse en aplicaciones, no directamente al programa original como sucede con los editores de recursos normales (no en tiempo de ejecución) sino sólo en memoria; The Customiser aplicará los cambios siempre que se arranquen dichos programas.

Esta herramienta ha llegado a convertirse en una pesadilla para muchos programadores que basan la protección de sus programas en un par de controles desactivados. Existen muchas maneras de activarlos; sin embargo, este método está al alcance incluso de todos los principiantes. Por esta razón se ha prevenido al lector varias veces contra los problemas relativos a este tipo de "protección" procurando orientarle a otros tipos de protección contra estos programas.

El uso de The Customiser resulta verdaderamente sencillo. Tras su arranque mostrará el siguiente cuadro de diálogo:

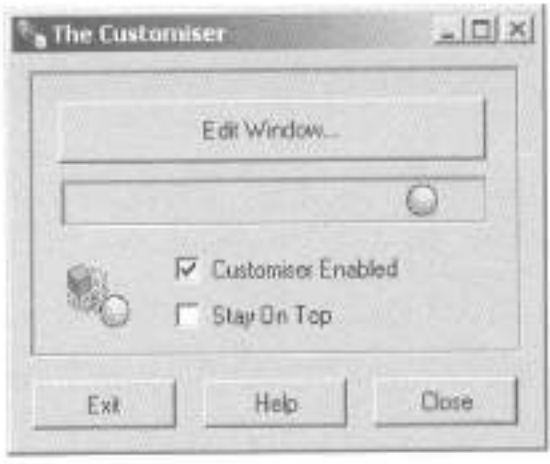

Figura 8-6. Apenas existen opciones en el cuadro de diálogo inicial de The Customiser

Las opciones del cuadro de diálogo inicial de The Customiser tan sólo permiten activarlo e indicar si debe permanecer siempre visible. Tras pulsar el botón "Edit Window", The Customiser mostrará el entorno de trabajo:

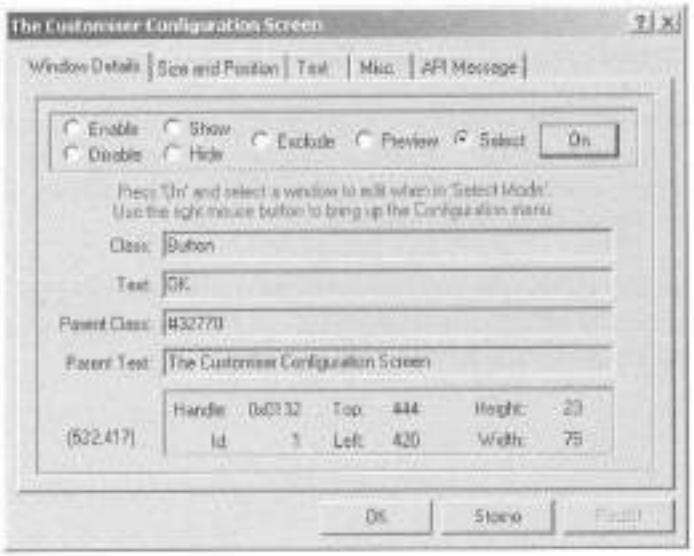

Figura 8-7. Opciones de la ventana de configuración

Al elegir una de las opciones de control ("Enable", "Show", etc.) y pulsar el botón "On", variará el curso del ratón y tras señalar el sitio elegido, se llevará a cabo la acción anterior. Al pulsar el botón "Off" no se podrá indicar ninguna ventana más.

Existen muchas otras funciones recogidas en las pestañas de otros menús: desplazamiento de objetos, modificaciones de texto, envío de mensajes Windows, etc. todas las pestañas anteriores contienen un menú para guardar las modificaciones.

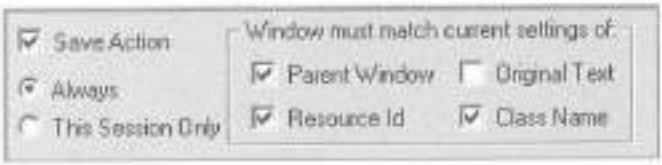

Figura 8-8. Cuadro de diálogo para "guardar" modificaciones hechas con The Customiser

Tras seleccionar la opción "Save Action", debe indicarse cómo se guardarán las modificaciones (con "Always" las modificaciones se cargarán siempre que se arranque y active The Customiser; con "This Session Only", los cambios sólo se aplicarán en esta sesión) y qué método de verificación se empleará para verificar la aplicación y la ventana en la que deban realizarse las modificaciones. No se olvide que éstas nunca se guardan directamente en el programa. The Customiser las aplica en memoria siempre que se arranque y active. Si se arrancase la aplicación sin que estuviese The Customiser activo, no tendría lugar ninguna modificación.

En el capítulo siguiente se incluirá un ejemplo de uso práctico de The Customiser. Tan sencillo resulta este programa que no debería suponer ningún obstáculo para nadie.

Naturalmente, The Customiser y programas semejantes no representan el único modo de, por ejemplo, activar y desactivar botones. Otro método muy común consiste en el uso de editores de recursos ordinarios (no en tiempo de ejecución), éstos sí guardan las modificaciones directamente en el programa.

Aunque útil, la enumeración aquí hecha no constituye ni de lejos una lista completa de los programas que utilizan los crackers. En el CD adjunto figuran varios programas de este tipo y pueden encontrarse muchos más en Internet.

CAPÍTULO 9

CRACKING DE ENTRENAMIENTO

Una vez leídos los capítulos anteriores, el lector se encuentra en disposición de practicar el "cracking". Como bien dice la consigna, "la mejor defensa es un buen ataque". Así que, en sintonía con tal principio, ¿qué mejor forma de comprobar la seguridad de una protección que atacarla? Éste será el objetivo precisamente de este capítulo. Mediante programas de entrenamiento, denominados "crackmes", diseñados con fines pedagógicos, el lector podrá aprender los fundamentos de las técnicas más actualizadas que los crackers utilizan hoy en dia para así aplicarlas con sus programas y comprobar su resistencia.

Se han elegido crackmes deliberadamente poco complicados para que se pueda apreciar rápidamente al menos los fundamentos del cracking como un todo. No habrán de considerarse estos ejemplos como un tipo de protección específica. Por el contrario, en la mayoría de los casos, muchos de los autores de crackmes cometen ciertos errores adrede para que hasta las personas sin experiencia puedan acceder a ellos. En otras ocasiones, se destacan deliberadamente en este entrenamiento los mismos errores pero de las protecciones actuales que sufren algunos programas con objeto de mostrar sus puntos débiles.

De estar interesado en ejemplos más complicados que ilustren las muchas técnicas y trucos de protección originales y con frecuencia excelentes, acúdase al CD adjunto y, por supuesto, a Internet.

CRUEHEAD - CRACKME V1.0

Con este ejemplo relativamente simple se ilustrará un mecanismo de protección basado en la introducción de un nombre a partir del cual queda generado el número de serie correcto. El algoritmo emplea una codificación muy sencilla basada en la función lógica XOR. A continuación se examina con detalle el procedimiento para demostrar con exactitud cómo descubrir la combinación correcta de número y nombre.

Arránquese el programa y elíjase el campo "Register" del menú "Help". En el cuadro de diálogo resultante rellénense en los campos "Name" y "Serial" con cualquiera valor. Se puede utilizar ZemoZ como valor de "Name".

Figura 9-1. Cuadro de diálogo con información estrictamente confidencial

Púlsese Ctrl+D para mostrar SoftICE. Ha de considerarse detenidamente cómo aproximarse lo más posible al algoritmo de protección. Un punto de entrada ideal al programa podría ser alguna llamada a una función API empleada para obtener información que se le suministrara –en nuestro caso, los dos campos "Name" y "Serial" –. Se prueba con la función API `GetDlgItemTextA`.

Defínase el punto de corte bpx a esta función API en SoftICE de la manera siguiente:

```
bpx GetDlgItemTextA
```

Vuélvase al programa (Ctrl+D) y púlsese el botón OK. SoftICE aparecerá inmediatamente al ser invocada la función API `GetDlgItemTextA` contra la que se definió el punto de corte. Ahora se sabe que el programa está leyendo el nombre y número introducidos y que por tanto invocará esta función API dos veces —la primera para leer el ítem "Name", y la segunda para leer el ítem "Serial"—. Púlsese F5 (o Ctrl+D) y SoftICE se detendrá en la segunda invocación a la función API. Al pulsar la tecla F11, se obtendrá

el código que invocó esta función, esto es, el código de nuestro programa. Bien es cierto que no sirve de mucha ayuda conocer el lugar desde el que se invocó la función API: habrá por tanto que pulsar F12 un par de veces para conocer el sitio desde el que se procesan los datos leídos. Púlsese F12 tantas veces como sean necesarias para retroceder al código del programa (se atravesará por un par de librerías). Éste será el código resultante:

```
:0040121E call USER32!DialogBoxParamA
:00401223 cmp eax,00000000
:00401226 je 004011E6
:00401228 push 0040218E    <--  nombre introducido -
                                Name
:0040122D call 0040137E    <--  obtención del resultado
                                de la operación XOR a
                                partir del nombre
                                introducido
:00401232 push eax    <--  guardando el resultado XOR
                           del nombre
:00401233 push 0040217E    <--  número introducido -
                                Serial
:00401238 call 004013D8    <--  cálculo del valor
                                correcto XOR según el
                                nombre introducido
:0040123D add esp,04    <--  alineamiento de pila
:00401240 pop eax
:00401241 cmp eax,ebx    <--  ¿valores idénticos?¿número
                              de serie introducido
                              correcto?
:00401243 jz 0040124C    <--  bifurcación = número
                              introducido correcto
```

Si se desea aceptar también una pareja de nombre y número incorrectos, deberá forzarse a la instrucción JZ en la dirección 00401243 a bifurcar. Con SoftICE se puede realizar esta operación acudiendo a la instrucción (pulsando repetidamente F10) y escribiendo el mandato siguiente:

```
rfl z
```

Este mandato cambiará el resultado de la anterior instrucción CMP a su contrario (modificando el indicador cero), lo que invertirá la lógica del algoritmo.

Ahora bien, se pretende descubrir el número correcto correspondiente al nombre introducido. Ello exige examinar las llamadas en las direcciones 0040137E y 004013D8.

```
:0040137E mov esi,[esp+04]    <--  nombre introducido
:00401382 push esi    <--  guardando en pila
:00401383 mov al,[esi]    <-- ESI = puntero al nombre
:00401385 test al,al    <--  ¿procesada la última letra
                               del nombre?
:00401387 jz 0040139C    <--  en caso afirmativo, final
:00401389 cmp al,41    <--  comparación con el valor
                              41h, i.e. A
:0040138B jb 004013AC    <--  bifurcación si El valor es
                              menor a A
:0040138D cmp al,5A    <--  comparación con el valor
                              5Ah, i.e. Z
:0040138F jae 00401394    <--  bifurcación, si a el
                             valor fuera mayor o igual a Z
:00401391 inc esi    <--  incremento del puntero a la
                             letra procesada del nombre
:00401392 jmp 00401383    <--  repitiendo, proceso del
                              nombre completo
:00401394 call 004013D2    <--  valor mayor o igual a Z
:00401399 inc esi    <--  incremento del puntero a la
                             letra procesada del nombre
:0040139A jmp 00401383    <--  repitiendo, proceso del
                              nombre completo
```

Esta parte del código comprueba que todas las letras vayan en mayúsculas (y también que el nombre contenga caracteres representados por un valor inferior al que representa la letra A). En caso contrario, los caracteres en minúsculas se convierten en mayúsculas invocando una función en la dirección 00401394. Esta función contiene la instrucción `SUB AL, 20h`.

Resulta muy probable que se produzca un error en el programa (se podría esperar una instrucción JA donde figura una instrucción JAE en la dirección 0040138F) al procesarse la letra "Z" y transformarse en el carácter ":". El valor 5Ah se transforma en 3Ah. Estas precisiones son pertinentes dado el nombre de registro elegido.

Estamos situados aún en la instrucción `CALL 0040137E`. Otra instrucción CALL figura en la dirección 0040139D – `CALL 004013C2`. Obsérvese lo que pasa con esta función:

```
:004013C2 xor edi,edi    <--  EDI = 0
:004013C4 xor ebx,ebx    <--  EBX = 0
:004013C6 mov bl,byte ptr [esi]    <--  ESI = puntero al
                                        nombre
:004013C8 test bl,bl    <--  ¿procesada la última letra
                             del nombre?
```

```
:004013CA jz 004013D1    <--  en caso afirmativo, final
:004013CC add edi,ebx    <--  suma de los valores de las
                              letras del nombre
:004013CE inc esi    <--  desplazamiento del puntero a
                          la siguiente letra del nombre
:004013CF jmp 004013C6    <--  próxima letra
```

Esta función añade los valores numéricos de todas las letras y guarda el resultado en el registro EDI. El nombre introducido ha quedado alterado debido a la rutina de transformación en letras mayúsculas de "ZemoZ" a ":EMO:". En este caso la suma de los caracteres del nombre será la siguiente:

```
:EMO:=3Ah +45h +4Dh +4Fh +3Ah =155h
```

A este valor se le aplica el operador XOR con el valor 5678h.

```
:004013A2 xor edi,00005678    <--  XOR
:004013A8 mov eax,edi    <--  el resultado se guarda en
                              el registro EAX
```

En este caso, el resultado de esta operación será 572Dh. El valor se almacenará entonces en el registro EAX y se comparará con el valor del registro EBX en la dirección 00401241. Este registro se completará con el segundo CALL en la dirección 00401238 - CALL 004013D8:

```
:004013D8 xor eax,eax    <--  EAX = 0
:004013DA xor edi,edi    <--  EDI = 0
:004013DC xor ebx,ebx    <--  EBX = 0
:004013DE mov esi,[esp+04]    <--  número introducido
:004013E2 mov al,0A    <--  AL = 10
:004013E4 mov bl,byte ptr [esi]    <--  ESI = puntero al
                                        número introducido
:004013E6 test bl,bl    <--  ¿ procesado el último
                             carácter del número?
:004013E8 jz 004013F5    <--  en caso afirmativo, final
:004013EA sub bl,30
:004013ED imul edi,eax
:004013F0 add edi,ebx    <--  guardando en EDI
:004013F2 inc esi    <--  desplazamiento del puntero al
                          siguiente carácter del número
:004013F3 jmp 004013E2    <--  proceso del carácter
                              siguiente del número
                              introducido
:004013F5 xor edi,00001234    <--  ¡¡aquí está!!
:004013FB mov ebx,edi    <--  guardando EDI en EBX (para
                              la siguiente comparación)
```

Este código sólo convertirá el número introducido en formato hexadecimal. No obstante, la información principal podrá hallarse en la dirección 004013F5. Al número convertido se le aplica el operador XOR con el valor 001234h y el resultado se almacena en el registro EBX.

Conviene recordar la instrucción en la dirección 004012241 - `CMP EAX, EBX`, quien decide la idoneidad del número de registro introducido. En este punto ya se sabe el valor correcto del registro EAX a partir de la CALL en la dirección 0040122D. El registro EBX será igual al número de serie introducido en formato hexadecimal al aplicarle la operación XOR con el valor 00001234h. Los registros EAX y EBX deberán ser iguales, lo que significa que la operación EAX XOR 00001234h coincidirá con el número de serie correcto.

EAX (en este caso) = 572Dh XOR 00001234h = 4519h = 17689

Por lo tanto, el número de serie correcto para el nombre ZemoZ será 17689.

CRUEHEAD - CRACKME V2.0

Figura 9-2. Introdúzcase la contraseña correcta

Este ejemplo resulta prácticamente idéntico al anterior, en este caso basta con introducir la contraseña en vez de la combinación de nombre y número de serie. Al igual que en el ejemplo anterior, aquí también se emplea la función lógica XOR en combinación con dos instrucciones CALL básicas. Se comparan al final los resultados de ambas instrucciones.

La estructura y construcción de este ejemplo es absolutamente idéntica a la anterior. La rutina principal del programa resulta muy semejante:

```
:00401228 push 0040217E     <--  contraseña introducida
:0040122D call 00401365     <--  idéntico al ejemplo
                                 anterior
:00401232 push 0040217E     <--  guardando el resultado
                                 de XOR con la contraseña
                                 introducida
:00401237 call 004013B8     <--  comparación de
                                 resultados
:0040123C add esp,04
:0040123F test cl,cl    <--  ¿valores iguales?
:00401241 jz 0040124A    <--  en caso afirmativo,
                              victoria
```

De la misma manera, la primera instrucción CALL convertirá a mayúsculas todos los caracteres que no lo sean y aplicará el operador XOR a su suma con un valor particular. En este caso el valor es distinto, concretamente "Messing_in_bytes", esto es, 4D 65 73 73 69 6E 67 5F 69 6E 5F 62 79 74 65 73h. El resultado de esta operación se guardará en la dirección 0040217E, muy fácil de obtener aun sin examinar las funciones individuales.

Al detenerse en la segunda instrucción CALL, se podrá observar que el resultado de la anterior operación se compara con el valor 1F 2C 37 36 3B 3D 28 19 3D 26 1A 31 2D 3B 37 3Eh. Queda claro que al aplicar el operador XOR a este valor con la clave codificada de "Messing_in_bytes", se obtendrá la contraseña correcta.

```
4D 65 73 73 69 6E 67 5F 69 6E 5F 62 79 74 65 73h XOR
1F 2C 37 36 3B 3D 28 19 3D 26 1A 31 2D 3B 37 3Eh
=RIDERSOFTHESTORM.
```

CRUEHEAD - CRACKME V3.0

Figura 9-3. Al menos puede saberse quién violó el programa

El tercer y último ejemplo de Cruehead emplea una protección basada en un fichero clave.

Como punto de entrada oportuno al programa, se utilizará la función API `CreateFileA` (con objeto de acceder al fichero, esto es, obtener su manejador). El programa buscará el fichero clave nada más arrancar, lo que exigirá habilitar SoftICE antes de arrancar el programa e introducir el siguiente mandato para establecer el punto de corte a la función anteriormente mencionada:

```
bpx CreateFileA
```

Tras arrancar el programa, SoftICE se mostrará. Púlsese F11 para acceder al código del programa. Se observará el código siguiente:

```
:00401028 push 004020D7
:0040102D call KERNEL32!CreateFileA
:00401032 cmp eax,-01   <--  ¿fichero hallado?
:00401035 jnz 00401043   <--  bifurcación = sí
```

Los parámetros de la función se almacenan en la pila en orden inverso, esto es, el último parámetro guardado será el primer parámetro de la función. El primer parámetro de la función `CreateFileA` será el nombre del fichero. La última instrucción `PUSH`, por lo tanto, señalará el nombre del fichero que se esté abriendo. Introdúzcase el mandato siguiente:

```
d 004020D7
```

El nombre del fichero —CRACKME3.KEY— figurará en la ventana de datos. Créese un fichero del mismo nombre y continúese el proceso:

```
:00401043 mov [004020F5],eax   <--  guardando el
                                    manejador del fichero
:00401048 mov eax,00000012   <--  número de bytes para
                                  leer del fichero
:0040104D mov ebx,00402008   <--  buffer de datos
:00401052 push 00   <--  parámetros de la función API
                         ReadFile
:00401054 push 004021A0
:00401059 push eax
:0040105A push ebx
:0040105B push dword ptr [004020F5]
:00401061 call KERNEL32!ReadFile   <-- invocación de la
                                   función API ReadFile
```

```
:00401066 cmp dword ptr [004021A0],12   <--  ¿12h bytes
                                             leídos?
:0040106D jnz 00401037   <--  en caso contrario, el
                              fichero no tiene el tamaño
                              correcto
```

El fichero se cargará en memoria en la dirección 00402008 (EBX) mediante la función API `ReadFile` para luego comprobar su tamaño. Resulta obvio a partir del código que el fichero debe tener el tamaño de 12h (18 bytes). Puesto que el fichero se editará repetidas veces en el futuro, resulta más apropiado emplear un editor hexadecimal.

Obsérvese ahora la siguiente instrucción CALL: `CALL 00401311`:

```
:00401311 xor ecx,ecx   <--  ECX = 0
:00401313 xor eax,eax   <--  EAX = 0
:00401315 mov esi,[esp+04]    <--  ESI señala al
                                   principio del buffer
                                   de datos
:00401319 mov bl,41   <--  BL = 41h
:0040131B mov al,[esi]    <--  AL = byte del fichero
                               crackme3.key
:0040131D xor al,bl   <--  XOR (descodificación)
:0040131F mov [esi],al    <--  guardando el resultado en
                               el buffer
:00401321 inc esi   <--  desplazamiento del puntero al
                         byte siguiente del fichero
:00401322 inc bl   <--  incrementando BL
:00401324 add dword ptr [004020F9],eax    <--  sencilla
                                               comprobación
                                               de integridad
:0040132A cmp al,00   <--  ¿es nulo el resultado de la
                           descodificación?
:0040132C jz 00401335   <--  bifurcación = final de la
                             descodificación
:0040132E inc cl   <--  cl++
:00401330 cmp bl,4F   <--  ¿BL = 4Fh? (4Fh - 41h = 14
                           bytes).
:00401333 jnz 0040131B   <--  bucle
:00401335 mov [00402149],ecx
```

Ésta es una sencilla rutina de descodificación, que aplica sucesivamente XORs a cada byte del fichero clave (nombre de usuario codificado) con un valor progresivamente mayor, comenzando en 41h y terminando en 4Fh. Lo que obviamente significa que el tamaño máximo de la parte del fichero clave descodificado será 4Fh-41h = 14 bytes. La

descodificación finalizará tan pronto como el resultado de la operación XOR sea cero o el valor de BL igual a 4Fh.

Para el nombre de usuario ZemoZ, la operación realizada será la siguiente:

ZemoZ = 5A 65 6D 6F 5Ah

XOR 41 42 43 44 45h

Result 1B 27 2E 2B 1Fh

Al rehusar utilizar los restantes bytes libres (14 – 5 bytes utilizados = 9 bytes) para guardar el nombre de usuario, resulta obligatorio que sea cero el resultado de la operación XOR. Al aplicar XOR con dos valores idénticos se obtiene dicho resultado. Al seguir al último valor utilizado 45h el valor 46h, será preciso añadir este último detrás del nombre de usuario.

El código del programa resultante:

```
:00401079 xor dword ptr [004020F9],12345678   <--
                          comprobación XOR 12345678h
:00401083 add esp,4
:00401086 push 00402008
:0040108B call 0040133C   <--  EAX = últimos cuatro
                              bytes del buffer
:00401090 add esp,4
:00401093 cmp eax,[004020F9]   <--  ¿comparación - EAX
                                  = comprobación de
                                 integridad?
:00401099 sete al   <--  si los valores fueran iguales,
                        entonces AL = 1
:0040109C push eax
:0040109D test al,al   <--  si AL = 1 no se ejecuta la
                           bifurcación siguiente
:0040109F jz 00401037   <--  bifurcación = no se ha
                            violado con éxito el
                           programa
```

Este código leerá los últimos cuatro bytes del fichero clave a partir del buffer y los comparará con el valor de la comprobación de integridad tras aplicarle la operación XOR con 12345678h. Por lo tanto, si aplicamos la operación XOR al valor de comprobación de integridad del nombre de usuario introducido con el valor 12345678h, se obtendrá el valor correcto de los últimos cuatro bytes:

Comprobación de integridad para el nombre de usuario ZemoZ:

```
5Ah +65h +6Dh +6Fh +5Ah =1F5h XOR 12345678h =1234578Dh
```

Resulta necesario escribir estos bytes en orden inverso (tras haberse obtenido del registro EAX, es preciso leerlos en el orden correcto). El fichero clave completo para el usuario ZemoZ será el siguiente:

```
00000000:1B 27 2E 2B-1F 46 00 00-00 00 00 00-00 00 8D 57

00000010:34 12
```

COSH - CRACKME1

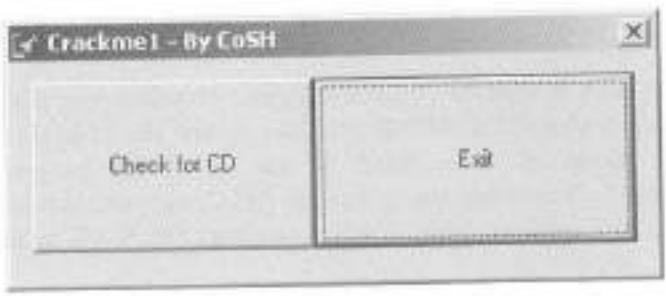

Figura 9-4. No hay muchos botones donde elegir

En este ejemplo clásico se utiliza la típica protección basada en la comprobación del CD.

En este contexto, la función API más conocida y empleada es `GetDriveTypeA`. Destínese a esta función un punto de corte mediante el mandato `bpx GetDriveTypeA` y púlsese entonces el botón "Check for CD". SoftICE mostrará lo siguiente:

```
:00401349 call KERNEL32!GetDriveTypeA
:0040134F cmp eax,03    <--  ¿DRIVE_FIXED?
:00401352 jz 00401392   <--  en caso afirmativo, probar
                             con otro disco
:00401354 lea eax,[ebp-18]
:00401357 push 00403058
:0040135C push eax
:0040135D lea eax,[ebp-20]
:00401360 push eax
:00401361 call 00401688
:00401366 mov eax,[eax]
:00401368 push ebx
```

```
:00401369 push ebx
:0040136A push ebx
:0040136B push ebx
:0040136C push 01
:0040136E push 80000000
:00401373 push eax    <--  CD_CHECK.DAT
:00401374 call KERNEL32!CreateFileA    <--  interesante
:0040137A cmp eax,-01    <--  ¿fichero hallado?
:0040137D lea ecx,[ebp-20]
:00401380 setz byte ptr [ebp-0D]    <--  si se detectó
                                         el fichero,
                                    entonces [EBP-0Dh] = 0
:00401384 call 0040169A
:00401389 cmp [ebp-0D],bl    <--  ¿[EBP-0Dh] = 0?
:0040138C jz 00401485    <--  bifurcación = comprobación
                              de CD anulada
```

En este caso, la función `GetDriveTypeA` se utiliza para detectar el primer disco fijo que no tenga el atributo `DRIVE_FIXED` —como el CD-ROM (el parámetro de la función que indica el disco sobre el que se realiza la comprobación cambia continuamente)—. A continuación, la función API `CreateFileA` intenta abrir el fichero CD_CHECK.DAT desde el disco, si no lo encontrara (EAX=-1), se mostraría un mensaje de error.

Figura 9-5. Aunque se haya fracasado se puede intentar de nuevo

La estructura del algoritmo es bastante típica y muy semejante a muchas comprobaciones de CD empleadas sobre todo en juegos. Basta con cambiar `JZ 00401485` en la dirección 0040138C a JMP. El fichero se habrá así "localizado" y aparecerá un mensaje informando de que esta protección se ha violado correctamente.

Figura 9-6. Mensaje de felicitación

Con un desensamblador todo queda más claro aún. Se puede observar en la lista de referencias de caracteres cómo la función API `GetDriveTypeA` va comprobando progresivamente todas las letras de los discos.

MEXELITE - CRACKME 4.0

Nada más arrancar, aparecerá el siguiente cuadro de diálogo:

Figura 9-7. Basta con ver este cuadro de diálogo para saber el tipo de protección empleada en este programa

Salta a la vista el tipo de protección que aquí se emplea: una combinación de nombre y número de serie. Introdúzcase cualquier nombre (que tenga al menos seis caracteres) y cualquier número, y definanse a continuación puntos de corte de sobra conocidos: `GetDlgItemTextA` y `GetWindowTextA`. Pronto se descubrirá que ninguno de los dos funciona. Obsérvese el programa con más detalle. El programa resulta bastante largo y las bien conocidas funciones API para obtener los contenidos de los campos del cuadro de diálogo no funcionan (tampoco funcionará aquí la función `MessageBoxA`)... Se habrá codificado el programa en Delphi.

Es de sobra conocido que los programas codificados en Delphi no emplean las funciones API `GetDlgItemTextA` y `GetWindowTextA` para obtener los contenidos de los elementos de control; por lo tanto, la siguiente posible solución sea establecer un punto de corte en la función API `Rmemcpy` (o bien emplear un desensamblador: se descubrirá la ubicación del algoritmo de comprobación a partir de la referencia de caracteres fácilmente). Definase un punto de corte a esta función, cuando aparezca SoftICE, púlsese F5 para leer el contenido del segundo campo (tanto el nombre como el número introducidos deben leerse) y deshabilítese el punto de corte.

Se obtendrá el código del programa pulsando un par de veces F12, o para ser más precisos, a la derecha de la sección del código, donde se compara el número de serie introducido con el correcto.

```
:0042DCA5 call 0041A228
:0042DCAA mov eax,[ebp-04]
:0042DCAD call 004065A8
:0042DCB2 mov [0042F760],eax   <--  guardando el número
                                    introducido
:0042DCB7 mov eax,[0042F758]   <--  EAX = número de
                                    serie correcto
:0042DCBC cmp eax,[0042F760]   <--  comparación entre
                                    el número de serie
                                    correcto y el
                                    introducido
:0042DCC2 jnz 0042DCDB   <--  bifurcación = combinación
                              incorrecta de nombre y/o
                              número
:0042DCC4 push 00
:0042DCC6 mov cx,[0042DD1C]
:0042DCCD mov dl,02
:0042DCCF mov eax,0042DDA0
:0042DCD4 call 0042CE40
:0042DCD9 jmp 0042DCF0   <--  gracias por registrarse
```

Aunque aquí no se pretenda, resulta fácil forzar al programa a aceptar también combinaciones incorrectas de nombre y número. Sin embargo, este procedimiento hallará un número de serie válido según el nombre introducido. El nombre empleado en este caso será "Mr.ZemoZ". A continuación se accederá a la dirección 0042DCBC donde se compara el número de serie correcto (en el registro EAX) con el número introducido para mostrar el contenido del registro EAX:

```
? eax
```

Se mostrará la siguiente información (válida para el nombre dado):

```
D67637DC     3598071772  (-696895524)
```

De manera que el número de serie correcto es –696895524.

IMMORTAL DESCENDANTS - CRACKME 8

Este excelente ejemplo codificado con Visual Basic 6.0 contiene muchos tipos de protecciones. Aquí sólo se estudiarán algunas de ellas.

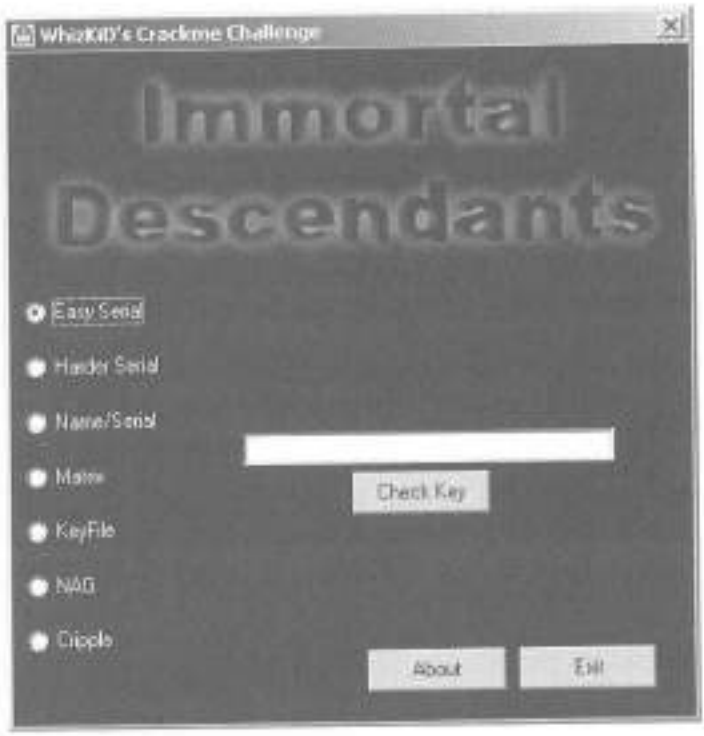

Figura 9-8. Diferentes tipos de protección

Lo primero que salta a la vista nada más arrancar el programa es una ventana de advertencia (en inglés, "NAG screen"). Se puede comenzar el cracking intentando suprimirla.

Figura 9-9. Ventana de advertencia que se pretende suprimir

En primer lugar, utilícese el cargador de símbolos para cargar las exportaciones de la librería de Visual Basic 6.0 —msvbvm60.dll (File->Load Exports..->file)—. Ahora sí se podrán definir puntos de corte a las diferentes funciones de esta librería.

Como ya se mencionó anteriormente, la función `rtcMsgBox` sí suele utilizar con Visual Basic para mostrar *MessageBox*. La definición del punto de corte a esta función será:

```
bpx msvbvm60!rtcmsgbox
```

SoftICE aparecerá nada más arrancar el programa. Púlsese F11 para volver al código del programa que invocó esta función. Aparecerá *MessageBox*, pulsando el botón "OK" se volverá a SoftICE.

```
:0040FDDA call [MSVBVM60!rtcMsgBox]
:0040FDE0 lea ecx,[ebp-1C]
```

A partir de aquí, con aplicar la instrucción NOP a CALL (sobrescribase la instrucción CALL con NOP tantas veces como sea preciso, 6 NOPs puesto que CALL tiene una longitud de 6 bytes), será suficiente para suprimir la molesta ventana de advertencia al principio de programa.

Se examinarán a continuación tareas individuales.

Easy Serial

Al ser ésta una sencilla protección basada en un número de serie, se intentará definir el punto de corte a la bien conocida función empleada para comparar secuencias de caracteres en Visual Basic, `vbaStrCmp`:

```
bpx msvbvm60!__vbaStrCmp
```

Introdúzcase cualquier número de serie y púlsese el botón "Check Key". Se podrá observar el siguiente código en SoftICE:

```
:0040E932 push eax   <--  parámetro de la función -
                         número introducido
:0040E933 push 0040BAB0   <--  parámetro de la función
                               - número correcto
:0040E938 call [MSVBVM60!__vbaStrCmp]   <-- invocación
                                             de la función
                                            de comparación
:0040E93E mov esi,eax
```

Así se puede comprobar lo potentes que son algunas funciones de comparación en Visual Basic: calcular el número correcto constituye una cuestión de segundos. En este

caso, la función `vbastrcmp`, toma las dos secuencias de caracteres comparados como parámetros.

De este modo, si se muestran los parámetros de la función, no sólo se podrá ver el número introducido sino también el número de serie correcto.

`d eax` mostrará sólo el número introducido; sin embargo, antes será necesario definir el punto de corte antes de invocar a la función ya que ésta modifica el valor del registro EAX.

`d 0040BABC` mostrará el número de serie correcto.

Los números siguientes aparecerán en la ventana de datos:

```
2.3.7.8.4.6.2.8.
3.5.6.2.6.7.....
```

Éste es el número de serie correcto en formato extenso. Éste es el formato empleado en todos los programas creados con Visual Basic: el desarrollador tendrá que acostumbrarse a él. Ignórense los espacios en blanco entre los números. El valor correcto es 23784628356267.

Ha resultado muy sencillo. Lo mejor de todo ello es que muchos otros programas utilizan exactamente el mismo método para comparar números de registro; el lector se sorprendería por la cantidad de programas que se puede violar de esta manera tan simple.

Harder Serial

No hay razón alguna que justifique denominar a este método "Harder Serial" (en castellano y respecto al método anterior: "más difícil"). Se puede emplear el mismo procedimiento anterior con idénticos resultados. La única diferencia radica en definir el punto de corte antes de invocar a la función `_vbaStrCmp` para evitar sobrescribir al registro EAX que señala al número de serie correcto.

```
d eax ->ADUJSDMD838707949BSOPEMNBD
```

Name/Serial

Este ejemplo ya es algo más difícil. Nada se obtendrá aquí con la función `_vbaStrCmp`, deberá considerarse otro punto de entrada distinto al programa. Se podrá definir el punto de corte a la bien conocida función `rtcMsgBox` quien dará cuenta del número de serie introducido incorrectamente. A continuación se podrá localizar desde

donde se invocó la función. También se puede establecer el punto de corte en la función API Hmemcpy. Ambos casos conducirán al mismo código:

```
:0040F322 mov eax,0000000A
:0040F327 mov ecx,80020004
:0040F32C mov [ebp-64],eax
:0040F32F mov [ebp-54],eax
:0040F332 mov eax,[ebp-18]    <--  d eax
:0040F335 mov [ebp-5C],ecx
:0040F338 mov [ebp-3C],eax
:0040F33B mov [ebp-4C],ecx
:0040F33E mov eax,00000008
:0040F343 lea edx,[ebp-74]
:0040F346 lea ecx,[ebp-34]
:0040F349 mov [ebp-18],ebx
:0040F34C mov [ebp-44],eax
:0040F34F mov dword ptr [ebp-6C],0040BC44
:0040F356 mov [ebp-74],eax
:0040F359 call [MSVBVM60!__vbaVarDup]
:0040F35F lea eax,[ebp-64]
:0040F362 lea ecx,[ebp-54]
:0040F365 push eax
:0040F366 lea edx,[ebp-44]
:0040F369 push ecx
:0040F36A push edx
:0040F36B push 10
:0040F36D jmp 0040F412 <--MessageBox - número de serie
                         o nombre introducido incorrecto
```

De introducirse el mandato d eax tras haber procesado la instrucción definida en la dirección 0040F332, se obtendrá algo muy semejante a lo siguiente en la ventana de datos:

```
I.D..C.r.a.c.k.
m.e..V.8...0...
B.y..W.h.i.z.K.
i.D. .......S...
```

Algo más adelante se podrá observar el número de serie correcto. Para el nombre aquí empleado —ZemoZ, el número de serie correcto es 272820394—.