Por Manu Roibal

Sistemas de protección

Como primer problema en la informática, debe citarse siempre al pirateo. Hay que tener en cuenta que es uno de los delitos más cometidos a nivel mundial en cualquier ámbito socio-cultural. El problema es muy grave. Por ello, nuestros programas deben disponer de sistemas de protección capaces de ofrecer una mínima resistencia al pirateo. En esta entrega se ofrece una solución software para la protección de los programas.

El sistema utiliza un sistema de autofirmado del ejecutable, es decir, el ejecutable obtiene información del sistema la primera vez que se ejecuta, y en posteriores ejecuciones compara esta información con la que obtiene nuevamente.

La función *IsPirate()* del fuente 1, implementada en la librería de funciones *FAST.lib*, comprueba si la variable pública *cFirma* contiene el valor *"@FAST@lib"*. Si es así, es que es la primera vez que se ejecuta el programa, y por lo tanto debe obtener información y escribirla en sí mismo. Para ello, llama a la función *IsProt()*. Lógicamente, al ser la primera ejecución, devuelve .F. a la alarma de pirateo. En posteriores ejecuciones se devolverá un valor lógico, que corresponderá a comparar la información del sistema con la anteriormente calculada.

```
// --- Fuente 1 ----------------------------------------------------

#define LenBuffer 32768


STATIC cFirma := "@FAST@lib"


FUNCTION IsPirate()
LOCAL lVal    := .T.
LOCAL cLetra := "@"
    IF cFirma == cLetra + "FAST" + cLetra + "lib"
      IsProt()
      lVal := .F.
    ENDIF
RETURN( lVal .AND. cFirma != DiskSerNum( 3 ) )



STATIC FUNCTION IsProt()
LOCAL nHandle
LOCAL cBuffer
LOCAL nLen
LOCAL nPos
LOCAL cletra := "@"
    nHandle := fOpen( cArgV( 0 ), 2 )
    nLen    := LenBuffer
    WHILE nLen == LenBuffer
      cBuffer := SPACE( LenBuffer )
      nLen    := fRead( nHandle, @cBuffer, LenBuffer )
      nPos    := At( cLetra + "FAST" + cLetra + "lib", cBuffer )
      IF nPos > 0
        fSeek( nHandle, nLen * ( -1 ), 1 )
        cBuffer := SubStr( cBuffer, 1, nPos - 1 ) + ;
                   DiskSerNum( 3 ) + ;
                   SubStr( cBuffer, nPos + 9 )
        fWrite( nHandle, cBuffer, nLen )
```

```
      ENDIF
    END
    fClose( nHandle )
RETURN NIL
```

Lo primero que hace la función *IsProt()* es abrir el programa en ejecución, es decir, abrir el fichero ejecutable que lo contiene, y lo hace en modo 2 (de lectura/escritura). Se puede contemplar el caso de que el programa esté protegido contra escritura forzando a que el fichero cambie sus atributos. El nombre del fichero no se conoce de antemano. Se usa la función *cArgV()* de *FAST.lib*, implementada mediante las indocumentadas *_argv* y *_argc*. Con ello, se consigue que el sistema siga funcionando aunque el usuario renombre el ejecutable.

Una vez hecho esto, se procede a localizar por el fichero el contenido inicial de la variable *cFirma,* que es "@FAST@lib" para modificarlo por la información del sistema y volver a grabar los nuevos datos en el fichero.

Algunos se estarán preguntando por qué se utiliza una variable *cLetra* que contiene el valor de @ en lugar de @ directamente. Muy sencillo, porque si en la comparación se pusiera @FAST@lib directamente, esta cadena sería encontrada varias veces en el ejecutable y no se sabría cuál debería cambiarse.

Como sistema para obtener información, se utiliza la función *DiskSerNum()* de *FAST.lib*, la cual devuelve el número de serie del disco duro. Este número es puesto por los programas formateadores desde la versión 4.0 de DOS.

En el fuente 2 se encuentra la función *DiskSerNum()*.

```
// --- Fuente 2 ----------------------------------------------------

CLIPPER DiskSerNum()
  {
  EXOREGS sRegs;
  BYTE    cTable[ 16 ];
  BYTE    cBuffer[ 9 ];
  LPBYTE  cBuffRM;
  LPBYTE  cBuffPM = _xalloclow( 25 );

  cBuffRM = ExoRealPtr( cBuffPM );
  sRegs.ds = FP_SEG( cBuffRM );
  sRegs.dx = FP_OFF( cBuffRM );
  sRegs.bx = _parni( 1 );
  sRegs.ax = 0x6900;
  ExoRMInterrupt( INT_DOS, & sRegs, & sRegs );
  _bcopy( cTable, (LPBYTE)"0123456789ABCDEF", 16 );
  cBuffer[ 0 ] = cTable[ cBuffPM[ 5 ] >> 4 ];
  cBuffer[ 1 ] = cTable[ cBuffPM[ 5 ] % 16 ];
  cBuffer[ 2 ] = cTable[ cBuffPM[ 4 ] >> 4 ];
  cBuffer[ 3 ] = cTable[ cBuffPM[ 4 ] % 16 ];
  cBuffer[ 4 ] = '-';
  cBuffer[ 5 ] = cTable[ cBuffPM[ 3 ] >> 4 ];
  cBuffer[ 6 ] = cTable[ cBuffPM[ 3 ] % 16 ];
  cBuffer[ 7 ] = cTable[ cBuffPM[ 2 ] >> 4 ];
  cBuffer[ 8 ] = cTable[ cBuffPM[ 2 ] % 16 ];
  _retclen( (LPSTR)cBuffer, 9 );
  _xfreelow( cBuffPM );
  }
```

Su uso es muy sencillo, aunque, nuevamente, la programación de los *buffers* para interrupciones en modo protegido complica algo las cosas. La función devuelve un *string* de 9 posiciones con el número de serie del disco duro.

Este método puede modificarse para guardar la información del primer día que se ejecutó y permitir la ejecución *n* días más, para obtener información sobre un servidor de red y de esa forma dar un licencia corporativa que funciona en un único servidor de red o cualquier otro método que pueda idearse. No obstante, como todo sistema de protección, tiene el problema de que el fichero debe estar asequible para lectura y escritura la primera vez que se ejecuta, lo cual no es mayor problema si el propio programador está presente la primera vez que se ejecuta el programa.

Y como tema estrella, me he reservado para el final un nuevo método de protección. Un sistema basado en la gestión de eventos que permite, por ejemplo, hacer que los programas funcionen durante cierto tiempo, que permite hacer pausas para mostrar carteles de versión *demo*... o cualquier otra cosa.

Está fundamentalmente basado en el sistema de eventos. Se trata de gestionar los eventos de tal forma que el motor del sistema nos envíe una señal cada vez que ocurra algo. De esta forma, tal como se ve en el fuente 3, podremos calcular el tiempo transcurrido de ejecución y actuar en consecuencia.

```
// --- Fuente 3 ----------------------------------------------------

#define DEMO_TIME  (DWORD)(18*60*2)

HIDE DWORD vPointerCont;
HIDE BYTE  vPointerFlag;


HIDE DWORD nTime( void )
  {
  WORD nDX, nCX;

  _AH = 0x0;
  geninterrupt(  0x1A );
  nCX = _CX;
  nDX = _DX;
  return( _fastShl( nCX, 16 )  + nDX );
  }


HIDE void _Interno_()
  {
  if ( ( nTime() >= vPointerCont + DEMO_TIME ||
         nTime() < vPointerCont ) && ( ! vPointerFlag ) )
    {
    vPointerFlag = TRUE;
    _gtScroll( 0, 0, _gtMaxRow(), _gtMaxCol(), 0, 0 );
    _gtWriteAt( 1, 0, "Tiempo excedido", 15 );
    _gtSetPos( 2, 0 );
    _gtSetCursor( 1 );
    _PutSym( _Get_Sym(  "_QUIT" ) );
    _tos++;
    _xDo( 0 );
    }
  }


CLIPPER IniciaProt()
  {
  vPointerFlag = FALSE;
  vPointerCont = nTime();
```

```
_evRegReceiverFunc( _Interno_, 0xFFFF );
}
```

Lo primero que se hace es inicializar la variable *vPointerFlag* a *FALSE* y tomar la hora del sistema en *vPointerCont* mediante *nTime()*. *vPointerFlag* lo usaremos para impedir anidamiento de llamadas a la función gestora de eventos. Mostrará *FALSE* o *TRUE,* indicando si ya estamos procesando un evento e impedir de esa forma que nuevos eventos sean procesados. La variable *vPointerCont* será usada para medir los tiempos que deseamos que sean controlados. La función *nTime()* hace una llamada a la interrupción *0x1A* para determinar el número de *ticks* de reloj que han pasado desde media noche. Recuerde que un segundo son aproximadamente 18.2 *ticks* de reloj. Aquí se usa la función *_fastShl()* para rotar un número de 2 bytes que se convierte automáticamente a 4 bytes (es decir, *long*) 16 posiciones hacia la izquierda y devolver un *long*.

La función *_Interno_* tan sólo tiene que medir el tiempo que quiere dejar pasar y comprobar que no se está ya procesando una interrupción. Si estas dos premisas se cumplen se borra la pantalla y se escribe el mensaje *tiempo excedido*. Posteriormente se busca el símbolo de la función *QUIT* de Clipper y se genera una llamada a esa función mediante *_xDo(0)*. También podríamos haber generado un error interno mediante *_ierror(),* en lugar de llamar a *QUIT*.

En este caso se ha decidido abortar la ejecución del programa una vez transcurrido el tiempo fijado de **18*60*2**, es decir, 2 minutos (*2 * 60 segundos * 18 ticks /sg*). Esto no tiene porqué ser siempre así. Una vez transcurrido el tiempo puede ponerse un cartel en la pantalla recordando que es una versión *demo* y volver a recalcular el valor de *vPointerCont* para sacar, nuevamente, el mensaje cuando vuelva a expirar el tiempo. Eso es ya misión del programador.