Ing. Inversa de troyanos II

Bueno aca los dejo a los que le interesen con la segunda parte
---

Bueno para empezar en la clase anterior quedamos pendientes con los saltos, así que será lo primero que veamos (salten esta parte los que se leyeron "ASM por CaoS ReptantE"), luego veremos algunas APIs de Windows y empezaremos a ver de refilón el debugger.

Instrucciones de salto

El listado de un programa consiste en una sucesión de instrucciones. Sin embargo a la hora de ejecutarlo, la ejecución del mismo no sigue el orden del listado, sino que, de acuerdo con distintas circunstancias y mediante las instrucciones de salto, se interrumpe la ejecución lineal del programa para continuar dicha ejecución en otro lugar del código. Las instrucciones de salto son básicamente de dos tipos: de salto condicional y de salto incondicional. En el primer tipo, la instrucción de salto suele ponerse después de una comparación y el programa decide si se efectúa o no el salto, según el estado de los flags (excepto en un caso, en el que el salto se efectúa si el valor del registro ECX o CX es cero). En el segundo, el salto se efectúa siempre.

Tipos de salto

Según la distancia a la que se efectúe el salto, estos se dividen en tres tipos: corto, cercano y lejano. También se dividen en absolutos o relativos, según como se exprese en la instrucción la dirección de destino del salto. Como veremos, los saltos cortos y cercanos son relativos, y los largos absolutos. En el salto corto, la dirección de destino se expresa mediante un byte, que va a continuación del código de la instrucción. Este byte contiene un número con signo que, sumado a la dirección de la instrucción siguiente a la del salto, nos da la dirección de destino del mismo. Como este número es con signo, podemos deducir que un salto corto sólo puede efectuarse a una distancia hacia adelante de 127 bytes y hacia atrás de 128. Veamos dos ejemplos de salto corto:

Código:
```
:004011E5 83FB05 cmp ebx, 00000005
:004011E8 7505 jne 004011EF
:004011EA C645002D mov [ebp+00], 2D
...
:004011EF 83FB09 cmp ebx, 00000009
:004011F2 72E2 jb 004011D6
:004011F4 58 pop eax
```
En el primer salto, la dirección de destino es la suma de la dirección de la instrucción siguiente y el desplazamiento: 4011EA+5=4011EF. En el segundo caso, E2 es un número negativo, por lo que la dirección de destino es la de la instrucción siguiente menos la equivalencia en positivo de E2 (1E): 4011F4-1E=4011D6. Estos cálculos, por supuesto, no los hace el programador, que simplemente dirige el salto a una etiqueta para que luego el compilador coloque los códigos correspondientes, pero me ha parecido que valía la pena explicarlo aquí.

- y aquí interrumpo yo de nuevo, nosotros NO trabajaremos desde un compilador, así

que no pondremos saltos a etiquetas si no a direcciones de memoria, pero no se preocupen que tampoco tendremos que hacer “cálculos de salto”

El salto cercano es básicamente lo mismo que el salto corto. La diferencia está en que la distancia a que se efectúa es mayor, y no se puede expresar en un byte, por lo que se dispone de cuatro bytes (en programas de 16 bits) que permiten saltos de 32767 bytes hacia adelante y 32768 hacia atrás o de ocho bytes (en programas de 32 bits) que permiten vertiginosos saltos de 2147483647 bytes hacia adelante y 21474836478 bytes hacia atrás.
Código:
```
:0040194F 0F8E96000000 jle 004019EB
:00401955 8D4C2404 lea ecx, dword ptr [esp+04]
...
:004019CB 0F8566FFFFFF jne 00401937
:004019D1 8D4C240C lea ecx, dword ptr [esp+0C]
```
En la primera instrucción, la dirección de destino es: 401955+96= 4019EB. En la segunda la dirección es: 4019D1-9A= 401937. Fijaos en que los bytes que indican el desplazamiento están escritos al revés y que 9A es la equivalencia en positivo de FFFFFF66.

Los saltos largos se utilizan cuando la instrucción de destino está en un segmento distinto al de la instrucción de salto.
Código:
```
:0003.0C28 2EFFA72D0C jmp word ptr cs:[bx+0C2D]
```
Saltos Condicionales

Las instrucciones de salto condicional sólo admiten los formatos de salto corto y salto cercano, por lo que su código está formado por el código de la instrucción más un byte (cb), un Word (cw) o un doubleword (cd) que determinan el desplazamiento del salto.

Aquí tenéis una relación de las instrucciones de salto condicional, desglosadas según el tipo de salto, en la que pueden apreciarse las equivalencias entre instrucciones de nombre distinto pero idéntica función.

Código:
```
        JA              Si es superior (CF=0 y ZF=0)
JNBE            Si no es inferior o igual (CF=0 y ZF=0)

        JAE             Si es superior o igual (CF=0)
JNB             Si no es inferior (CF=0)
JNC             Si no hay acarreo (CF=0)

        JNA             Si no es superior (CF=1 o ZF=1)
JBE             es inferior o igual (CF=1 o ZF=1)

        JNAE            Si no es superior o igual (CF=1)
JB              Si es inferior (CF=1)
JC              Si hay acarreo (CF=1)

        JG              Si es mayor (ZF=0 y SF=OF)
JNLE            Si no es menor o igual (ZF=0 y SF=OF)

        JGE             Si es mayor o igual (SF=OF)
JNL             Si no es menor (SF=OF)
```

```
        JNG             Si no es mayor (ZF=1 o SF<>OF)
JLE             Si es menor o igual (ZF=1 o SF<>OF)

        JNGE            Si no es mayor o igual (SF<>OF)
JL              Si es menor (SF<>OF)

        JE              Si es igual (ZF=1)
JZ              Si es cero (ZF=1)

        JNE             Si no es igual (ZF=0)
JNZ             Si no es cero (ZF=0)

        JO              Si hay desbordamiento (OF=1)

        JNO             Si no hay desbordamiento (OF=0)

        JP              Si hay paridad (PF=1)
JPE             Si es paridad par (PF=1)

        JNP             Si no hay paridad (PF=0)
JPO             Si es paridad impar (PF=0)

        JS              Si es signo negativo (SF=1)

        JNS             Si no es signo negativo (SF=0)

        JCXZ            Si CX=0 //esta instrucción solo es interpretada
JECXZ           Si ECX=0        //por procesadores INTEL
```

Salto incondicional

Es un salto que no está sujeto a ninguna condición, es decir, que se efectúa siempre. Hay una sola instrucción: **jmp**. Esta instrucción admite los tres tipos de salto: corto, cercano y lejano. Creo que con unos ejemplos será suficiente:
Código:

```
:004011DA EB30 jmp 0040120C
:00402B35 E9F8000000 jmp 00402C32
:0001.02B4 E94D01 jmp 0404
:0003.0C28 2EFFA72D0C jmp word ptr cs:[bx+0C2D]
```

Ahora que hemos visto todos los saltos habidos y por haber pasemos a otro tema, las API.

Como se dijo en la clase 1, son instrucciones preprogramadas que le facilitan la vida al programador, pero para poder utilizarlas, se debe indicar en el ejecutable cuales son las que van a utilizar, y en que librería se encuentran (aunque no lo crean, hay apis repetidas en Kernel32.dll y Ntdll.dll), pero además de esto también hay que pasarle unos parámetros o argumentos, para que la función sepa lo que va a hacer.

La forma de pasarle estos argumentos a las APIs es metiéndolos en la pila (push) antes de llamar a la API.
Código:

```
Push 40         //tipo de MessageBox
Push 402000     //puntero del titulo del MessageBox
Push 402010             //puntero del texto del MessageBox
Push 0          //manejador del MessageBox
Call MessageBoxA
```

Como pueden ver en este ejemplo, se introducen los argumentos a la pila antes de la llamada a la API, los “punteros” son las direcciones que contienen los verdaderos

argumentos.

Para cada API los argumentos son diferentes. Y aquí les dejo algunas APIs, lo que hacen y los argumentos que necesitan

MessageBoxA

Crea, despliega, y maneja una caja de mensaje. La caja de mensaje contiene un mensaje definido por el programa y un título, más cualquier combinación de iconos y botones predefinida.
Código:

```
    Int MessageBox(
     HWND  hWnd,                                    // el manipulador
de ventana padre
     LPCTSTR  lpText,                             // la dirección del
texto dentro de la caja de mensaje
     LPCTSTR  lpCaption,                        // la dirección del
título de la caja de mensaje
     UINT  uType                                    // el estilo de
caja de mensaje
    );
```

Returns
El valor de retorno es cero si no hay bastante memoria para crear la caja de mensaje. Si la función tiene éxito, el valor del retorno es uno de lo siguiente valores devueltos por la caja de diálogo:

IDABORT, IDCANCEL, IDIGNORE, IDNO, IDOK, IDRETRY, IDYES

Si una caja de mensaje tiene un botón de Cancelación, la función devuelve el valor IDCANCEL si la tecla ESC es apretada o si de botón de Cancelación es seleccionado. Si la caja de mensaje no tiene ningún botón de Cancelación, apretar ESC no tiene efecto.

CreateFileA

La función CreateFile crea, abre, o trunca un archivo, pipe (*), fuente de comunicaciones, dispositivo de disco, o consola. Devuelve un manipulador que puede usarse para acceder al objeto. También puede abrir y puede devolver un manipulador para un directorio.
Código:

```
     HANDLE CreateFile(
      LPCTSTR  lpFileName,                          // la dirección de
nombre del archivo
      DWORD  dwDesiredAccess,          // el modo de acceso (leer-
escribir)
      DWORD  dwShareMode,                           // modo share
      LPSECURITY_ATTRIBUTES  lpSecurityAttributes,// la dirección de
descriptor de seguridad
      DWORD  dwCreationDistribution,           // cómo crear
      DWORD  dwFlagsAndAttributes,             // los atributos del
archivo
      HANDLE  hTemplateFile                          // el
manipulador de archivo con atributos para copiar
     );
```

Returns
Si la función tiene éxito, el valor del retorno es un manipulador abierto del archivo

especificado. Si el archivo especificado existe antes de función llamada y dwCreationDistribution es CREATE_ALWAYS u OPEN_ALWAYS, una llamada a GetLastError devuelve ERROR_ALREADY_EXISTS (aunque la función ha tenido éxito). Si el archivo no existe antes de la llamada, GetLastError devuelve cero.

ReadFileA

La función de ReadFile lee datos de un archivo, comienza en la posición indicada por el indicador del archivo. Después de que la operación de lectura se ha completado, el indicador del archivo es ajustado realmente por el número de bytes leídos, a menos que el manipulador de archivos sea creado con el atributo superpuesto. Si el manipulador de archivos es creado para entrada y salida superpuesta (I/O), la aplicación debe ajustar la posición del indicador del archivo después de la operación de lectura.
Código:

```
     BOOL ReadFile(
      HANDLE  hFile,                                    // el
manipulador de archivo para leer
      LPVOID  lpBuffer,                               // la dirección de
buffer que recibe los datos
      DWORD  nNumberOfBytesToRead,           // el número de bytes
para leer
      LPDWORD  lpNumberOfBytesRead,          // la dirección de número
de lectura de los bytes
      LPOVERLAPPED  lpOverlapped             // la dirección de
estructura para el dato
     );
```

Returns
Si la función tiene éxito, el valor del retorno es TRUE. Si el valor de retorno es que TRUE y el número de lectura de los bytes es cero, el indicador del archivo estaba más allá del extremo actual del archivo en el momento de la operación de lectura.

Si la función falla, el valor de retorno es FALSE. Para conseguir información extendida del error, llama a GetLastError.
WriteFile

La función WriteFile escribe datos a un archivo y está diseñado para la operación sincrónica y asíncrona. La función comienza escribiendo datos al archivo en la posición indicado por el indicador del archivo. Después de que la operación de escritura se ha completado, el indicador del archivo es ajustado realmente por el número de bytes escritos, excepto cuando el archivo es abierto con FILE_FLAG_OVERLAPPED. Si el manipulador de archivos se creara para la entrada y salida solapada (I/O), la aplicación debe ajustar la posición del indicador del archivo después de que la operación de escritura es terminada.
Código:

```
    BOOL WriteFile(
      HANDLE  hFile,                                  // el
manipulador de archivo para escribir
      LPCVOID  lpBuffer,                            // la dirección de
datos para escribir al archivo
      DWORD  nNumberOfBytesToWrite,          // el número de bytes a
escribir
      LPDWORD  lpNumberOfBytesWritten,   // la dirección del número de
bytes escritos
```

```
      LPOVERLAPPED  lpOverlapped            // la direc. De estructura
necesaria para I/O solapada
     );
```

Returns
Si la función tiene éxito, el valor de retorno es TRUE.

Si la función falla, el valor del retorno es FALSE. Para conseguir información extendida del error, llama a GetLastError.

RegCreateKeyExA

La función RegCreateKeyEx crea la clave especificada. Si la clave ya existe en el registro, la función lo abre.
Código:

```
    LONG RegCreateKeyEx (
      HKEY  hKey,                                  // el manipulador
de una clave abierta
      LPCTSTR  lpszSubKey,                    // la dirección del
nombre de subclave
      DWORD  dwReserved,                      // reservado
      LPTSTR  lpszClass,                      // la dirección de clase
de string
      DWORD  fdwOptions,                      // las opciones
especiales de flag
      REGSAM  samDesired,                     // acceso de seguridad
deseado
      LPSECURITY_ATTRIBUTES  lpSecurityAttributes, // la dirección de
estructura de seguridad de clave
      PHKEY  phkResult,                       // la dirección de buffer
para el manipulador abierto
      LPDWORD  lpdwDisposition          // la dirección de valor de
disposición del buffer
     );
```

Returns
Si la función tiene éxito, el valor de retorno es ERROR_SUCCESS.

Si la función falla, el valor de retorno es un valor de error.
(la forma de llamar a estas apis en ASM es hacer push a los parámetros, poniendo los que están de últimos, de primeros –por ejemplo, el parámetro lpdwDispsition de la ultima API explicada, es al que primero se debe hacer push-)

Eso es todo en cuanto a APIs por hoy, ahora pasaremos al debugger que será en lo que nos enfoquemos de aquí en adelante…

Empecemos con las partes del mismo, pero una imagen vale más que mil palabras

Ahora vamos a ver que función tiene cada parte…

En el desensamblado, es donde trabajaremos la mayor parte del tiempo y es donde se nos muestran las instrucciones a ejecutar.

La barra de información, es donde se nos muestra la información acerca de cada instrucción (valores con los que trabaja, desde donde es llamada X subrutina, etc.)

El dump, es donde se nos muestra los valores hexa de los segmentos de la memoria

La pila o Stack, es eso la pila, es donde se guardan datos provisionalmente, y donde se pasan los parámetros a las API.

Los registros nos muestran el valor de los mismos, con posibilidad de modificarlos

Los botones de ventanas, nos muestran las distintas ventanas al presionarlos:

- los conceptos no explicados, serán vistos sobre la marcha en esta u otras clases.

El botón “L” nos muestra el Log, que si lo vemos apenas al cargar cualquier ejecutable, veremos que nos dice que cargo los plugins y el ejecutable de forma satisfactoria, esta ventana nos será de especial atención, pues será aquí donde se nos muestre los parámetros que se le pasan a las API cuando son llamadas, de esta forma sabremos todo lo que hace nuestra victima (esto después de haber hecho ciertas cosillas que veremos en las próximas clases)

El botón “E” nos muestra los módulos que se han cargado en memoria, es decir el ejecutables, las dll, los ocx y cualquier otro modulo que necesite para funcionar (pudiendo llegar a ser otro ejecutable, pues aunque no lo crean, los ejecutables también pueden exportar funciones, siendo así otros ejecutables necesitaran cargar este ejecutable a la memoria para poder utilizar sus funciones exportadas)

El Botón “M” nos muestra la memoria, es decir los módulos cargados, sus secciones, trozos de memoria reservados a ciertos módulos por la llamada a la función VirtualAlloc, etc.

El botón “T” nos muestra los threads o hilos creados, permitiéndonos “matarlos”, suspenderlos, etc.

El botón “W” nos muestra las ventanas creadas por nuestro ejecutable.

El botón “H” nos muestra los handles o manejadores.

El botón “C” nos muestra la “CPU” es decir la ventana que esta en la imagen donde esta la pila, el dump, etc.

El botón “/” nos muestra los parches o cambios hechos a mano, (o debería, pero yo nunca he visto nada en esa ventana)

El botón “K” nos muestra el “Call Stack” es decir las subrutinas en las que hemos entrado, o mejor dicho las subrutinas en las que estamos dentro.

El Botón “B” nos muestra los breackpoints que hemos puesto, y el estado de los mismos.

El botón “R” nos muestra las referencias, pero para que nos muestre algo debemos primero buscarlas, en esta ventana se nos muestran las referencias a la instrucción que especifiquemos, las referencias de texto dentro del ejecutable, las llamadas intermodulares, etc.

El botón “…” nos muestra el “run trace” es decir, aquí quedan logueadas las instrucciones ejecutadas cuando se tracea. Para esto debemos “abrir el run trace” esto se hace en el menú DEBUG > Open or Clear Run Trace.

Y el botón “S” al poner el Mouse encima de el nos dice “Show Source” que en español será algo así como “mostrar fuente”, pero nunca he sabido para que es esta ventana…

Los botones de acción, los explicare según el orden en el que aparezcan en la imagen (por ejemplo “el tercer botón hace esto, esto y lo otro) así que estad atentos con la imagen.

El primer botón, es el que tiene la forma de un botón de play (►) y lo que hace es correr el programa, y su tecla de acceso rápido es F9.

El segundo botón, es el botón de pause (▌▌) y hace lo que es obvio pausa o detiene la ejecución del programa..

El tercer botón, de forma de flecha doblada a la derecha, (algo parecido a esto ┗►□) es el botón “step into” y lo que hace es que ejecuta instrucción por instrucción, entrando en las calls (se entenderá mejor en la practica) su tecla de acceso rápido es F7.

El cuarto botón, es una flecha que medio se asoma por arriba, (este si no se como expresarlo T_T) es el botón “step over” y hace lo mismo que el anterior, ejecuta instrucción por instrucción, pero con la diferencia de que con este no entramos a los calls, si no que se ejecutan las subrutinas completas. Su tecla de acceso rápido el F8.

El quinto botón, es una flecha zigzagueante, es el botón trace into. Lo que hace es eso tracear (ejecutar instrucción por instrucción, pero continuamente, es decir sin parar, al

“abrir el Run trace” quedara logueado ahí, cada instrucción ejecutada. Cuando nosotros ejecutaos paso a paso con F8 o F7 también se le llama tracear) entrando en las calls. Su combinación de acceso rápido es Ctrl. + F7.

El sexto botón, es el botón de “trace over” que hace lo mismo que el anterior, pero sin entrar en las calls. Su combinación de acceso rápido es Ctrl. +F8.

Y el séptimo botón, es el botón es el botón “till return” y lo que haces es ayudarnos a salir de una subrutina, es decir, que ejecuta las instrucciones hasta encontrar una instrucción RETN y ahí para. Su combinación de acceso rápido es Ctrl. + F9.

Bueno, esto es todo por ahora… disculpen (de nuevo) lo largo del post, se que deben estar MUY fastidiados después de leer todo esto (si me pagaran por hacer post largos, me haría millonario con esta serie de clases), pero ahora ya están listos para la practica, que empezaremos desde la próxima clase. Ojala les haya gustado y es pero que **shaddy** no me saque la madre por cualquier error cometido xD.