Prologo

Como introducción a esta serie de artículos que van a comenzar a publicarse en lalista de crakslatinos, los autores Ricardo Narvaja y mR gANDALF, hacen una breve entrada de lo que pretende ser esta colección de tutoriales sobre el craking.

Este el comienzo de algo nuevo. No les voy a mentir si les digo que este es un proyecto ambicioso que hemos comenzado mi amigo mR gANDALF y yo. Creo que lo que vamos a realizar es algo único en su especie.

Muchos dirán que ya hicimos un curso el cual existe y esta completito. Dicho curso, al igual que este, estarán en mi FTP del cual se dan los datos.

La diferencia entre este curso y el anterior, es que pensamos hacer de este algo más grande, con tutes míos y de mR gANDALF y con los mejores tutes de otros autores, mencionando quien es,

así como traducciones al castellano de tutes en ingles hechas por nosotros de los autores que consideremos los mejores sobre el tema y también con los mejores tutes del curso anterior corregidos, actualizados, modificados, revisados y puestos lindos para que tengan una mejor lectura y comprensión.

Así que será esto distinto. También, cada varias lecciones pensamos hacer como una pequeña auto evaluación, así cada uno puede ver si hasta allí llegó entendiendo todo y está en condiciones de seguir adelante, realizando algún ejercicio que pondremos cuya respuesta saldrá en el comienzo de la siguiente parte, para que exista un método de auto-verificación de

Hace un año, aproximadamente, un amigo me paso un programa en un CD-R que después de instalar te pedía un número de serie. Como yo no conocía tal número, lógicamente no pude utilizarlo. Cuando hablé con mi amigo le dije: "Oye, me tienes que dar el serial del programa por que si no, no peta". "Tienes el crack en la carpeta de instalación", me respondió él.

¿El crack.?...yo no sabía... Puede parecer extraño a quién lea esto explicarse como es posible que ya en pleno siglo XXI alguien no conociera lo que es un crack.

Pues bien, yo mismo no lo sabía. Meses después mi hermano menor trató de ayudarme a buscar algo para que me funcionara una demo de 30 días de un programa que ya le había tomado cariño. Tecleo http://astalavista.box.sk/ y el nombre del programa, eligió un *patcher* acorde con la versión y en pocos minutos mi demo recobró la vida. Definitivamente esto llamó poderosamente mi atención. ¿Cómo era posible que un programa que funcionaba de una determinada manera, de pronto empezase a funcionar de otra? . El primer impulso que tuve fue el de recuperar todos esos CD`s llenos de *shareware* que muchas veces desee poseer. Por debajo rumiaba la curiosidad de saber hasta que punto astalavista era capaz de hacerlo. Me quedé asombrado,

conocimientos. Quiero decir a los que recién se inician que no tengan miedo, que no se asusten. Por allí van a leer sobre ensamblador y sobre lenguajes de programación y mi temor es que piensen que para empezar a crackear hay que ser un experto en estas cosas. Yo les digo que no es necesario para comenzar a crackear saber ensamblador, salvo unas mínimas ideas que son muy sencillas y que uno aprende de a poquito, sin necesidad de ser un experto. Es bueno decir que cuando yo comencé a crackear no tenía ni idea de ensamblador y ya había crackeado como 50 programas. Allí fue cuando profundice un poco mas sobre el tema y es a día de hoy que no soy un experto en ello, pero voy tratando de mejorar y seguir aprendiendo, que es algo que nunca termina.

En este curso, a diferencia del anterior, van a tener muy buenos tutes de otros autores sobre ensamblador que serán material de consulta y estudio, pero siempre creo y creeré fervientemente que es muy bueno saber ensamblador para crackear pero es mejor es dotado de

sentido común, persistencia, practicar mucho y crackear todo lo que se nos cruce en el camino.

Aunque al principio fallemos no importa. Si algún programa nos venció, lo dejamos archivado y después de un tiempo cuando se incrementen nuestros conocimientos, lo volvemos a intentar, y quizás entonces tengamos mas suerte.

Estas ideas son lo que se me ocurrió a mí cuando comencé y es lo que sigue ocurriéndoseme ahora, con todo lo que me falta por aprender.

miles de cracks con cada consulta. Patches, loaders, serials, keygens...Un universo nuevo para un concepto que aún era desconocido para mí: *la ingeniería inversa.* Como dice el *gigante* Black Fénix en su Introducción al Cracking que presentamos en el primer capítulo, hay una evolución en esto cuyo primer peldaño refleja ese ansia acaparadora que han bautizado los anglosajones con el termino *lammer*. Estos son unos personajes muy comunes cuya motivación principal es la de no pagar por el software que consumen, así de sencillo. Según Black Fénix, algunos evolucionan por el camino de la curiosidad hacia algo más constructivo y se denominan con el termino *newbie.*

Este era la ruta que yo recorría, sin saberlo muy bien, por la senda que ya había trazado *el mago*. El primer tutorial que cayo en mis manos fue "My First Lesson" de PCQuest. No me pregunten dónde lo encontré por que ya no lo recuerdo bien y todo rastro fue borrado de mi disco duro durante un mal encuentro con un *black list* (harcoded que el programa reconoce produciendo graves consecuencias para el newbie atrevido). Posteriormente creo que leí la introducción de Yodacker, que seguramente escribo mal y cuya lectura fue quizás la más definitiva de cuantas haya realizado sobre este apasionante tema. Este escrito dormitó durante muchas noches en una carpeta de mi escritorio (una carpeta *real* en un *escritorio* de madera). Internamente se debatía una lucha entre la curiosidad y la prudencia. ¿Sería capaz de introducirme en este *oscuro arte,* como muchas veces había leído llamar al craking, tan solo

Por otro lado, quisiera agradecer a mi amigo mR gANDALF, cuya habilidad para crackear y realizar tutes ya se vio en el curso anterior, su sentido de la organización y el orden, algo de lo cual yo carezco, ya que soy todo un caos y desorden. Todo lo ordenadito que sea este curso será mérito de mR gANDALF, que ya verán es en ese sentido muy diferente a mí y creo que por eso nos complementaremos bien.

Ya veremos si les gusta. No me queda mas que incitarlos a que comiencen este curso de cracking y que lean mucho, investiguen y practiquen, que no es nada del otro mundo y que todos cuando empezamos de cero éramos unos simples curiosos que nos pico un bichito y solo sabíamos que teníamos ganas de ir metiéndonos poco a poco en el apasionante mundo del cracking.

Ricardo Narvaja, Abril 2002

para poder llegar a comprender esos sesudos tutoriales que corrían por la red? ¿Sería un esfuerzo baldío?. Poco a poco fueron cayendo uno tras otro cientos de tutoriales. La página de karpoff fue definitiva. *La Meca* del conocimiento sobre ingeniería inversa en castellano, el lugar donde se producen colaboraciones para traducir textos, para sumar nuevos tutoriales con cada entrega del maestro y guardián del saber, esa persona imprescindible sin la que tantas cosas jamás llegarían hasta nosotros. Las lecciones de AESOFT, que aún dudamos de incluir, fueron el bautismo de fuego. Después vinieron las lecciones del curso de Ricardo, el apuntarme a la lista de crackslatinos y el encontrar en él no solo a un maestro, sino a un amigo en quien poder confiar y con quien poder compartir momentos inolvidables como "aquel preciso instante en el que modificando el último byte, allá... a lo lejos... en el horizonte de tu monitor... si te fijas bien, puedes observar como resopla...Mobby Dick " (Crack el Destripador).

mR gANDALF, Abril 2002

Índice

Capítulo 1

Guía General de Craking bajo Windows

Black Fenix

Entre las mejores páginas sobre ingeniería inversa se encuentra Reversed Minds, de Black Fénix, que aunque no se actualizará ya más ha dejado para los amantes del cracking una excelente colección de tutoriales y una de las mejores introducciones del tema que jamás se han escrito. Por ello, queremos abrir así esta nueva colección de artículos para los que se inician al cracking y para todos aquellos que tengan la curiosidad de acercarse a él..
mR gANDALF

Bienvenidos a lo que pretende ser una guía general de cracking bajo Windows. Esta guía trata de introducir a aquellas personas sin conocimientos sobre el tema en el apasionante mundo del hacking/cracking. Si usted ya es un experto, felicidades, no le hace falta leer esto, aunque siempre podrá aprender algo nuevo. Un consejo para todos los interesados en este mundillo y en general el mundillo de la informática: aprende inglés, sabiendo inglés podrás acceder a las fuentes de información más actuales, obteniendo así el mejor nivel sobre el tema.
Para poder entender lo aquí explicado es necesario un mínimo conocimiento del lenguaje ensamblador y de Windows, si no se conoce el lenguaje ensamblador, mejor que busques algún tutorial en la web[1]. En cuanto a Windows, no es necesario ser un experto programador de este sistema (sin operativo porque no lo es).

¿Qué es el Hacking/Cracking?

Se podría definir como un proceso manual o automático en el cual un sistema de seguridad que protege un software/hardware es "burlado". El termino hacking se aplica cuando este proceso tiene lugar en un sistema remoto, es decir se asalta el sistema de un ordenador ajeno, pudiendo así acceder a información confidencial pudiendo llegar a destruirla, alterarla o copiarla. Existen los Hackers blancos, que en principio sus intenciones no son maléficas y tan sólo buscan fallos de seguridad en el sistema y los Hackers negros o crackers de redes, los cuales pueden destruir los datos, copiarlos,

alterarlos o hacen lo que les plazca cuando llegan a obtener todos los privilegios de acceso.

Alguien definió una escala bastante absurda en la que digamos se muestra la evolución de un hacker. Muchos quedan estancados en alguna fase de la escala y otros ni tan siquiera logran pasar de la primera etapa.

Lamer -> Wannabe -> Newbie -> Hacker -> Gurú o Elite -> Wizard

Por supuesto llegar a hacker no es cuestión de dos días y llegar a Wizard es algo que casi nadie logra. Entre la élite podemos encontrar a gente bastante joven pero los Wizards suelen ser ya mayores y debido a su inmensa experiencia, tanto en la vida misma como en la informática, poseen un nivel que ni ellos mismos podrían concretar donde acaba. En mi opinión es difícil distinguir a la élite de los wizards y creo que los Wizards necesitan tener un carácter especial como persona, por lo que no todo el mundo puede llegar aquí. Alguien que se jacte de ser élite y vacile con ello nunca llegará a Wizard, al contrario, descenderá rápidamente a niveles inferiores.

El termino cracking es aplicable cuando se burla el sistema de protección de algún software que normalmente esta protegido contra copia u otro tipo de protección similar (versiones limitadas por tiempo, por un número de serie, mochilas, etc) pudiendo así obtener una copia funcional de un software protegido. Cabe diferenciar entre crackers de redes y crackers de sistemas de protección de software. Algunos "gurus" del cracking también lo definen como un estado diferente de la mente, otros como una manera de expresar su arte, incluso algunos la definen como una ciencia (y resulta cierto).

¿Qué necesito para entrar en combate?

He aquí la pregunta del millón. Pues para empezar es necesaria una mínima base de ensamblador por lo que si no tienes ni idea te recomiendo que consultes algún libro o busques algún tutorial en la web[2]. Mira el apartado de links de mi página, allí encontrarás algo que podrá ayudarte. Estar bien equipado es la clave para obtener el éxito en combate, he aquí un listado con las herramientas que necesitaremos, a parte de claro está, paciencia:

SoftIce for Windows 9X/NT: Es un debugger[3] para Windows, el mejor en su clase. Esta es la base de nuestro ejército, sin él no haremos nada ante un enemigo bien preparado. Se recomienda profundizar mucho con él y aprendérselo bien.

W32dasm: Esta aplicación es un desensamblador/debugger para Windows. Con él abriremos al enemigo en canal y examinaremos su estructura y código. Es muy sencillo de usar.

IDA Pro: Otro desensamblador. En algunos casos el W32dasm no funcionará correctamente por lo que usaremos este otro que funciona mucho mejor aunque es más complicado. Necesario si no se usa W32dasm.

Ultra-Edit 32: Editor hexadecimal de lujo con el modificaremos el código del software que así lo precise. Puedes usar otro editor pero te recomiendo este por ser muy versátil.

Filemon: Este programita nos permitirá monitorizar que archivos son abiertos por la aplicación/es que le digamos y así saber con que ficheros trabaja el enemigo. Este arma sólo será necesaria en algunos casos ;)

Monitor del API de Windows: Programa que nos permitirá monitorizar todas las funciones del API de Windows que están siendo ejecutadas por determinada aplicación. No es imprescindible pero tampoco es inútil ;)

RegEdit: Este programa nos permitirá monitorizar todos los accesos al registro de Windows. Así sabremos donde guarda la información el enemigo;). Opcional.

Win32 Programmer's Reference: Una guía que describe todas las funciones del API de Windows. Imprescindible si no se domina bien el API, cosa más que probable (por lo menos en mi caso).

Te recomiendo que te familiarices con estas tools puesto que si no tiene un mínimo conocimiento del arma[4] te puede salir el tiro por la culata (pérdidas de datos etc. ;)). Puedes buscar estas herramientas en **www.astalavista.box.sk**o en mis links.

Obviamente no he listado todas las herramientas de las cuales podemos sacar partido ya que sería una lista interminable, por eso, siempre deberemos visitar algunas de las siguientes páginas para obtener más "**armamento**". Todo armamento que podamos conseguir nos puede ahorrar horas de trabajo, aunque no es bueno abusar de las armas, ya que hay veces que hay que luchar sin ellas y si no estamos acostumbrados podremos fracasar[5].

- **http://www.suddendischarge.com**
- **http://www.hackersclub.com**
- **http://crknotez.cjb.net**
- **http://fravia.org**
- **http://protools.cjb.net**

Sistemas de protección más usuales

Es crucial que antes de comenzar el combate conozcamos a qué nos enfrentamos. Por ello, antes de empezar una sesión de cracking, deberemos realizar una exploración preliminar del enemigo - utilizar nuestros satélites espía[6];**-)** - . Las principales características que deberemos conocer dependerán del tipo de protección que este siendo usado, pudiendo ser un sistema de protección combinado (+ de un tipo de protección) lo cual puede llegar a complicar el asunto. Una vez detectado el sistema de protección

comprobaremos las posibles defensas del enemigo. He aquí las defensas que deberemos tener más en cuenta:

Anti-Debugging: El software usa técnicas para detectar nuestra presencia y evitar así poder ser trazado[7]. Normalmente intentará detectar la presencia de un debugger y parar la ejecución del software mientras este presente.

Encriptación/Compresion de datos: El software usa técnicas que "ocultan" el verdadero código del programa hasta que se ejecuta el programa, inutilizando así cualquier intento de desensamblado del código.

Antes de entrar en combate deberemos conocer al enemigo. Una vez conocido deberemos seleccionar el armamento adecuado y estaremos listos. Este paso es importante ya que nos evitará muchos quebraderos de cabeza si lo realizamos.

Como llegar al núcleo de la protección

Cuando estamos trazando un determinado software podemos utilizar varias técnicas para llegar antes a la parte de código en la cual sospechamos que puede haber parte del sistema de protección. Aquí describiré algunos de los métodos más usados.

A lo retro: Este método se basa en trazar el programa hacia atrás, es decir, dejar que el sistema de protección se active y parar la ejecución[8] justo después (cuando el software nos avise con un mensaje de error). A partir de ese instante trazaremos hacia atrás (esto lo haremos examinando el código) buscando un salto que nos aleje o nos aproxime de la función que muestra el mensaje: típicamente esta función suele ser una de las siguientes:

- **MessageBoxA, MessageBox,**
- **DialogBoxParam, DialogBoxParamA**

Por Predicción: Este método se utiliza cuando se sospecha que una determinada función del API de Windows esta siendo usada para el funcionamiento del sistema de protección. Se pone un breakpoint[9] (BPX) en el SoftIce a la función que se sospecha esta siendo usada y se carga/continua con la ejecución del software. A partir de ahí se continua trazando normalmente hasta llegar al punto clave. Muy usada cuando se quiere buscar la función que pinta un banner molesto o una pantalla nag de inicio o cuando se conoce el sistema de protección que esta usando el enemigo;).

Por referencia a una cadena conocida: Este método se usa cuando el sistema de protección muestra un mensaje de error o un mensaje dentro de un cuadro de diálogo. Se copia el mensaje de error, se desensambla el archivo con el W32dasm, se busca dicha cadena en la lista de referencias a cadenas y se hace doble click en esta, se apuntan

las direcciones donde hay una posible referencia y se examina el código que hay alrededor buscando un salto que nos aleje de la referencia a dicha cadena. (similar al trazado a lo retro). Se comprueba que sucede si se invierte el salto o preferiblemente se examinan las llamadas previas al salto si las hay, ya que estas pueden ser las llamadas que comprueban si todo está en orden (fecha/mochila/numero de serie etc.).

Por búsqueda de cadenas: Se utiliza cuando se sospecha que una determinada cadena de caracteres está siendo utilizada/o y no se encuentra donde debería estar por el método de referencias. Esto es debido a que el software puede almacenar ciertas cadenas en partes reservadas al código del programa, para evitar que estas sean directamente visibles en el desensamblador. Se busca con un editor hexadecimal en el archivo que se sospecha contiene la cadena, o se busca la cadena en memoria con el SoftIce una vez ejecutado el programa y antes de que se ejecute el sistema de protección. Si se encuentra en memoria se pone un BPM sobre esta para interrumpir al programa antes de realizar cualquier cálculo con ella;).

Por búsqueda de una secuencia de códigos de operación: Se utiliza cuando se sospecha que una determinada secuencia de órdenes en assembler está siendo usada por el sistema de protección / defensa. Conocida la cadena secuencia de códigos/bytes a buscar se realiza su búsqueda con editor hexadecimal y se opera según sea el caso. Muy usada para la detección de trucos Anti-debugging.

Sistemas de protección por tiempo

Los sistemas de protección por tiempo pueden operar de distintas formas:

1. El software **comprueba si han transcurrido X días** desde la instalación de sí mismo, y si es así el software procede a su salida inmediata o en el peor de los casos a su desinstalación automática. Durante la salida / desinstalación del software este puede mostrar algún mensaje informando al usuario del hecho en cuestión. Puede que el software vuelva a funcionar durante X días si se vuelve a instalar. Ej: The evaluation period has expired...bla bla. Trans: Periodo de evaluación a terminado....bla bla.

2. El software **comprueba si ha llegado a una fecha limite**, si es así procede de la misma manera que en el caso anterior. La diferencia está en que el software dejará de funcionar a partir de una fecha determinada y no funcionará si se vuelve a instalar.

Para burlar estos sistemas de protección deberemos tener en cuenta los siguientes datos:

1. El software sabe **cuando se instaló por primera vez,** por lo que deberá guardar esta información en algún lugar, probablemente en algún archivo propio (usaremos Filemon) o en el registro de Windows (usaremos RegMon).

2. El software debe **comprobar la fecha actual,** es decir la fecha en el momento en el que el programa está siendo ejecutado, **y hacer los correspondientes cálculos** para comprobar los días que han pasado o si es la fecha límite. Para esto dispone de las funciones de fecha / tiempo del API de Windows siguientes:

   - VOID **GetSystemTime**(LPSYSTEMTIME lpSystemTime);
   - VOID **GetLocalTime**(LPSYSTEMTIME lpSystemTime);

   Estas dos funciones pasan un puntero a una estructura de tipo **SYSTEMTIME** con la siguiente definición:

```
typedef struct SYSTEMTIME {
        WORD            wYear;
        WORD            wMonth;
        WORD            wDayOfWeek;
        WORD            wDay;
        WORD            wHour;
        WORD            wMinute;
        WORD            wSecond;
        WORD            wMilliseconds;
} SYSTEMTIME;
```

   La función **GetTickCount** retorna el número de ticks transcurridos desde el arranque del ordenador, puede ser usada para controlar el tiempo de ejecución de una rutina, con esto podrían saber si hay alguien trazando el código:)

```
DWORD GetTickCount(VOID)
```

   Algunos programas que usan las MFC (Microsoft Foundation Classes) utilizan la siguiente función para calcular fechas:

```
double difftime( time_t timer1, time_t timer0
);  //
Devuelve la diferencia entre dos tiempos en
segundos.
```

   El PaintShop Pro 5 por ejemplo usa esta función. Para más información sobre estas funciones ver la guía de referencia del API de Windows[10].

Con esto en mente ya podemos deducir cual será el método a seguir para burlar el sistema: poner breakpoints (BPX) en cada una de estas funciones y trazar paso a paso a partir de donde se ejecutó la función que provocó el BPX. También podemos usar el método de las referencias a cadenas. Existen casos en los que estos métodos no nos servirán de mucho ayuda o simplemente no nos facilitarán la tarea, estos casos se dan cuando el software está protegido con algún sistema de protección comercial por tiempo como el TimeLock o VBox. Este tipos de sistemas se basan en DLL's externas, por lo que será más práctico el desensamblado y trazado de estas. En algunos casos simplemente bastará parchear alguna función de la DLL para haber terminado con el sistema de protección, con la ventaja de que todos los programas basados en esa protección estarán automáticamente desprotegidos (habremos crackeado cientos de aplicaciones que ni siquiera conocemos:-)).
Para más detalles puedes leer mis tutoriales sobre cracking números **1**,**5** y **6**.

Sistemas de protección por banner o nags

Estos sistemas no son propiamente un sistema de protección, más bien son sistemas para molestar al usuario del software y recordarle que adquiera el programa original, los "banners" se utilizan mucho en programas de visualización o de retoque de fotografías, se trata de textos o imágenes que tapan parcialmente el trabajo que estamos viendo/haciendo, impidiéndonos su "correcta" visualización. Los "nags" son pantallas/cuadros de diálogo que aparecen al inicio o al final de la aplicación y están activos hasta que el usuario pulsa un determinado botón o hasta que se completa una cuenta atrás. Por lo tanto para que exista un banner o un nag, este debe ser mostrado de alguna manera, ahora sólo nos queda saber como se muestran los nags, y como lo hacen los banners.

1. **Técnicas para hacer frente a los nags:** Lo primero sería identificar el tipo de "nag", es un cuadro de dialogo ? es un cuadro de mensaje ? O es otro tipo?. Esto lo podremos averiguar con facilidad si echamos una mirada al estilo del "nag". Si hay 2 o menos botones, es probable que sea un cuadro de mensaje, si no hay botones o hay más de dos, probablemente será un cuadro de diálogo. Otra pista es mirar las opciones que hay escritas en los botones, si dicen "Continuar", "Aceptar", "Cancelar", "OK", es probable que sea un cuadro de mensaje, Otra posibilidad es fijarnos en los iconos del cuadro, si vemos un signo de admiración, de información o una cruz en un circulo rojo, seguramente se trate de un cuadro de mensaje.

   Anticipándonos a la creación del cuadro de diálogo / mensaje.
   Poniendo BPX en las siguientes funciones:

   - **CreateDialogIndirectParamA /**
   - **CreateDialogIndirectParam**
   - **CreateDialogParamA /**
   - **CreateDialogParamW**

- **DialogBox**
- **DialogBoxIndirect**
- **DialogBoxParam /**
- **DialogBoxParamA /**
- **DialogBoxParamW**
- **EndDialog**
- **MessageBeep**
- **MessageBoxA / MessageBoxW**
- **MessageBoxExA /**
- **MssageBoxExW**
- **MessageBoxIndirect /**
- **MessageBoxIndirectA /**
- **MessageBoxIndirectW**

Estas funciones son las que Windows puede utilizar para crear un cuadro de diálogo / mensaje, las cinco primeras son las más utilizadas, las cuatro últimas son sólo para cuadros de mensaje. Para más información sobre estas funciones ver la guía de referencia del API de Windows.

Por referencia a una cadena conocida Una vez localizada la llamada al cuadro de diálogo/mensaje, procederemos a su eliminación, para ello lo más habitual será eliminar la llamada al cuadro de diálogo/mensaje mediante NOP o instrucciones similares (INC EAX, DEC EAX etc.). Es posible que después de la llamada al cuadro de diálogo, este retorne un valor el cual la aplicación deberá procesar para así continuar su ejecución, por eso deberemos tener en cuenta este valor (el valor correcto para que la aplicación continúe) y hacer que el registro/memoria que contenía dicho valor, siga valiendo igual al eliminar la llamada al cuadro de diálogo. Esto lo podremos lograr aprovechando el espacio que nos quede al eliminar la llamada, o simplemente cambiando el salto que compruebe este valor.

Técnicas avanzadas de eliminación de nags: En ciertas ocasiones, no podremos eliminar la llamada al cuadro de diálogo / mensaje debido a que esta llamada está codificada como una llamada indirecta (p.ej: call [ESI+EBX]) o cuando la aplicación usa la misma llamada para que se procese más de un cuadro de diálogo. Es de imaginar que si eliminamos esta llamada, también estaremos eliminando otras posibles llamadas a otros cuadros de diálogo que no sean el molesto "nag", como un cuadro de diálogo de configuración etc. La solución no es muy compleja: **todos los cuadros de diálogo** que la aplicación pueda generar **deben estar identificados por un número ID, o cadena** que los identifica inequívocamente, sabemos que

todas las funciones de generación de cuadros de diálogos necesitan que este ID o cadena sea pasado/a como un argumento a la función que no podemos eliminar, por lo que podremos averiguar el ID del cuadro de diálogo que nos molesta mirando los push previos a la llamada. Una vez tengamos dicho ID podremos desensamblar la aplicación con el **W32dasm**, buscar el ID en la lista y hacer doble click sobre este para el **W32dasm** nos lleve a las posibles referencias a este ID, con esto podremos intentar eliminar / sustituir el cuadro de diálogo. Para más información puedes ver mi tutorial número 8 en el se trata un caso práctico.

2. Técnicas para hacer frente a los banners[11]:

Si es un simple banner de texto podremos poner un BPX en las siguientes funciones. Tenemos que ir con mucho cuidado con estas funciones ya que son usadas por el GDI de Windows para pintar los textos de los controles (botones, listas, etc.)

- **DrawText**
- **DrawTextEx**
- **ExtTextOut**
- **TextOut**

Si el banner es una imagen:

- **BitBlt**
- **MaskBlt**
- **PatBlt**

Para más información sobre estas funciones ver la guía de referencia del API de Windows. Para más información puedes ver mis tutoriales número 8 y 5 en ellos se tratan dos casos prácticos.

Sistemas de protección, CD Checks

Estos sistemas de protección comprueban que el CD del software se encuentra en la unidad de CD-ROM cada vez que el software se ejecuta. Se utilizan para evitar lo que se llaman CD-Rips (versiones recortadas de la versión original del CD, sin intros, sin audio etc) usados por los piratas para introducir más de un juego en un sólo CD. Estos sistemas no evitan que el CD-ROM pueda ser duplicado, lo que significa que si introducimos una copia, el software funcionará correctamente. Yo casi ni los considero un sistema de protección, más bien son un sistema de fastidiar al usuario obligándole a introducir el CD, aunque el software no necesite ningún dato del CD para funcionar.
Personalmente me he encontrado con dos sistemas de este tipo, en el primer sistema, el software identifica el CD-ROM mediante la etiqueta que este posee, si el CD tiene la etiqueta esperada, el software continua su ejecución. El segundo método, se basa en comprobar si existe un determinado archivo

dentro del CD-ROM, si el archivo existe, se continua con la ejecución. La dificultad de estos sistemas de protección se ve aumentada cuando el software si que verdaderamente necesita los datos que hay en el CD para poder continuar (intros, pistas de audio etc.).

1. Técnicas utilizadas para hacer frente a los CD checks:

Anticipándonos a la detección de la unidad de CD: Poniendo BPX en las siguientes funciones:

```
AUINT GetDriveType(
   LPCTSTR lpRootPathName //
dirección del cámino raíz
);
```

Esta función es usada por windows para detectar el tipo de unidad pasada como cadena en lpRootPathName, si retorna 5, la unidad es un CD-ROM, a partir de aquí iremos trazando poco a poco hasta llegar a la comprobación del CD Recuerda que el BPX será BPX **GetDriveTypeA**, ya que es una función de 32 bits.

Anticipándonos a la detección de la etiqueta de la unidad de CD: Poniendo un BPX en las siguientes funciones

```
BOOL GetVolumeInformationA(
   LPCTSTR lpRootPathName,   //
dirección del directorio raíz del sistema
de archivos system
   LPTSTR lpVolumeNameBuffer,   //
dirección del nombre del volumen
   DWORD nVolumeNameSize,   //
longitud del buffer
lpVolumeNameBuffer
   LPDWORD
lpVolumeSerialNumber,   // dirección
del numero de serie del volumen
   LPDWORD
lpMaximumComponentLength,   //
dirección de la máxima longitud del
nombre del archivo del sistema
   LPDWORD lpFileSystemFlags,   //
dirección de los flags del sistema de
archivos
   LPTSTR
lpFileSystemNameBuffer,   //
dirección del nombre del archivo del
sistema
   DWORD
nFileSystemNameSize     // longitud del
```

buffer
);
GetVolumeInformationW
Estas funciones obtienen un puntero a una cadena conteniendo la etiqueta del volumen de la unidad especificada, a partir de aquí es fácil buscar donde se compara con la etiqueta válida. La función **GetVolumeInformationW** obtiene la cadena en formato wide-char (o eso creo).

Para más detalle sobre este tipo de protecciones puedes echar un vistazo a mi tutorial número 6 te aconsejo que te mires también la guía de referencia del API de Windows.

Sistemas de protección anticopia

Los sistemas de anticopia no se exponen aquí ya que no serán tratados en este manual, al menos inicialmente. Los interesados en este tema visiten la página de Reversed Minds donde encontrarán explicación detallada y referencias sobre este tipo de protección.

[1] En los capítulos 2 y 3 se trata este tema.

[2] Algunas nociones básicas para empezar se dan en los capítulos 2 y 3.

[3] Debugger o depurador. Herramienta que se utiliza para localizar los fallos de un programa, ósea, para depurarlo. También es la herramienta fundamental del cracker para el que los fallos del programa lo constituyen sus medios de protección.

[4] Las herramientas básicas del cracking se exponen partir del capítulo 4, y todas ellas podrán obtenerse en los FTP`s *asociados* al curso.

[5] Algunas de las webs clásicas del cracking. No son las únicas, aunque sí algunas de las mejores. Para mi gusto faltan muchas incluso mejores que estas.

[6] *Satélites espía*: Se refiere a los llamados *analizadores de archivos*, que son programas que nos dicen con que fue compilado un archivo. Se da detallada información de esto en el Volumen II.

[7] Trazando: Del inglés Trace, seguir el rastro. Algunos decimos tracear. Significa correr un fragmento de la aplicación con un depurador, generalmente el SoftIce, que encontrareis abreviado como Sice.

[8] *Parar el programa*: Se refiere al método de depuración con Sice.

[9] Breakpoint: Punto de ruptura. En Sice un breakpoint detiene la ejecución del programa, la congela (Ice), y nos muestra el valor de los registro y de la memoria justo en ese preciso instante, así como las instrucciones que acaban de ejecutarse y las que vienen justo a continuación.

[10] Manual de Referencia de las API de Windows. Imprescindible para el cracker medio. Viene en inglés y en formato de ayuda, aunque muchas de estas se encuentran traducidas en las crackers notes en español (Txeli) y actualmente se esta llevando a cabo un importante esfuerzo por parte de algunos para traducir la mayoría de estas (ver la pagina de Karpoff).

[11] Banner: Del inglés, estandarte. Se refiere a esos molestos avisos que nos parece al correr una demo indicándonos que debemos registrarnos. A menudo lo hacen aparentemente al azar, surgiendo de improviso.

Capítulo 2

Ensamblador I: Conceptos Básicos.

Wintermute

De entre los múltiples tutoriales de ensamblador que circulan por la red uno de los mejores, por su contenido y claridad, es sin duda alguna el del ***"Curso de programación de virus"*** *de Wintermute, de la pagina underground chatsubo bar. Tengo que recomendar fuertemente la lectura de estos dos tutoriales de Asm que para el newbie serán sin duda los más provechosos de cuantos lea y para el "cracker" medio serán también una agradable sorpresa si no los conocía. Debo agradecer el descubrimiento de esta fascinante página a numit_or, cuyos continuos trabajos sobre ingeniería inversa merecen el máximo respeto.* **mR gANDALF**

Programar en ensamblador no es fácil, pero cuando se le coge el tranquillo es extremadamente gratificante; estás hablando directamente 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.

Algunos conceptos previos

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.

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 después. 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,??.**

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.

La orden MOV y el acceso a memoria

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

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.

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.

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).

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.

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 directamente. 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).

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 (sí "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:

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.

OR

El OR lógico también opera bit a bit, poniendo el resultado a 1 sí 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.

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

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

Operaciones aritméticas

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

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 (sí 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.

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 sí 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 hiciéramos 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.

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 *1234567887654321h*, EAX contendría *87654321h* y EDX *12345678h*.

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.

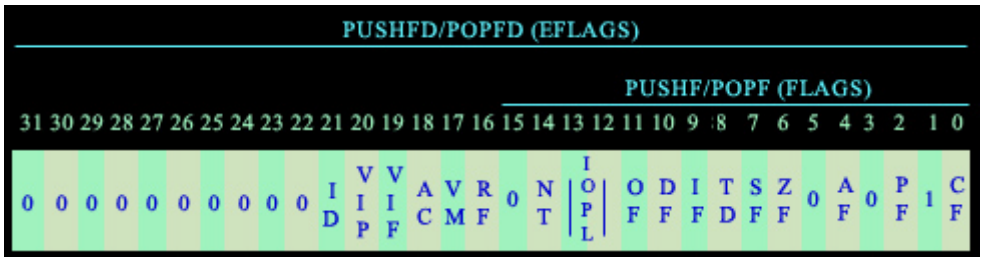
INC y DEC

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.

Registro de estado (FLAGS) e instrucciones de comparación

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 sí 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** sí 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 llamábamos 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 directamente con la instrucción **CLI** (Clear Interrupts), mientras que **STI** (Set Interrupts) lo activa), se prohíbe 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 directamente 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".

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 directamente 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).

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.

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.

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 directamente el estado de los flags (como pueda ser el de paridad). Entre ellos incluimos 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

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.

La pila

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 directamente 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 exactamente 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 sobrescritos. Si nuestra orden 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). Exactamente, estas dos instrucciones afectan a *EAX, EBX, ECX, EDX, EBP, ESI y EDI.*

Subrutinas

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 directamente 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
```

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.

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 directamente 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 3

Ensamblador II

Wintermute

Esta segunda parte sobre el lenguaje ensamblador versa sobre algunos aspectos más avanzados, como son las API`s , las dll`s, las instrucciones de cadena y otros temas interesantes. **mR gANDALF**

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.

Interrupciones y API

Introducción

La API es 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 usan las 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, en el sistema operativo Linux.

Interrupciones en MS-Dos[i]

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 todavía
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 (creación, lectura/escritura, borrado...) y directorios.

La Int80h y Linux

En Linux pasa tres cuartas de lo mismo, pero todas las funciones del sistema están reunidas bajo una sola 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 **INT 80h**, 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 orden **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 habrá 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)

DLL's y llamadas en Windows[ii]

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).

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):

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)
```

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).

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.
%
```

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.

CLC/STC

Se trata de dos instrucciones que manejan directamente 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).

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.

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 sustituida 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 orden **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 orden **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.

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).

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.

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.

Otras instrucciones interesantes[iii]

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).

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 = "ineI' 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

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...).

Introducción al coprocesador[iv] matemático

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.

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.

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.

[i] Ya prácticamente no se ven aplicaciones escritas para MS-DOS, por lo que esta sección es meramente anecdótica.

[ii] En muchas ocasiones durante nuestras sesiones de crackeo necesitaremos saber que API`s hacen una determinada tarea, p.ej, mostrar un cartelito diciendo que la contraseña introducida es incorrecta, e incluso *como* lo hacen. Para documentarnos en esta materia disponemos de las Crackers Notes, traducidas ya al español, con las API`s de uso más frecuente, y la WIN32 SDK Reference help, mucho más completa, cuya traducción se está llevando a cabo actualmente (para los que quieran colaborar busquen en la sección traducciones de la página de Karpoff).

[iii] Muchas de estas instrucciones las veremos solo ocasionalmente. No es necesario conocerlas todas, ni mucho menos saber como funcionan. Solo cuando se nos presente una duda al respecto de estas instrucciones menos comunes será interesante regresar aquí para documentarnos sobre su uso. La mejor *Biblia* sobre ensamblador y la referencia última que pueden consultar en caso de apuro es la monumental obra The Art of Assembly (www.webster.cs.ucr.edu/).

[iv] En palabras del propio autor, de lectura "opcional", ya que a estas alturas más de uno puede estar *agonizando...*

Capítulo 4

The Tools of the Trade I: W32Dasm 8.x

Urbanik

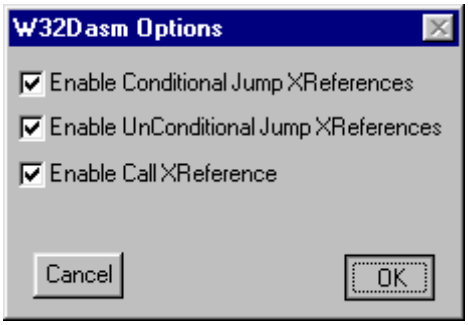
Wdasm es, generalmente, la primera herramienta que conoce el Newbie, la que le da sus primeros éxitos y la que probablemente seguirá utilizando mucho tiempo por su facilidad y rapidez de uso. Lo que se presenta en este capitulo es el tutorial que viene con la ayuda del programa, y que muestra como es una sesión de depuración de Calc.exe (versión para Win95). La traducción corre a cargo de ***noseman****. La revisión y composición del texto fue hecha por* ***mR gANDALF.***

Este tutorial le llevara paso a paso a través de una típica sesión de debugging usando el W32Dasm ver 8.xx . El mejor camino para aprender es practicando, pero es recomendable que usted siga este tutorial.

NOTA: Su computador debe estar a una RESOLUCION de 1024 x 768 x 256 colores como mínimo para proveer una ventana lo suficientemente confortable para correr el debugger del W32Dasm.

Empezando: Desensamblando calc.exe

1.1 Abra el w32dasm.
1.2 Seleccione "**Disassembler Options**" del menú de disassembler. Aparecerá una caja de dialogo.



Contiene las opciones Call Cross References, Conditional Jumps, y UnConditional Jumps. Activando o desactivando estas opciones podrá incluir referencias en cada línea de código del texto desensamblado que es referenciado por una instrucción de Call o Jump.

Nota: activando estas opciones podrá sustancialmente incrementar el tiempo que toma para desensamblar un archivo.

1.3 Seleccione el botón "**Open File to Dissassemble**" del menú de disassembler ó presione el botón  de la barra de herramientas.

1.4 Seleccione el archivo "calc.exe" que encontrara en el directorio de windows[1] y presione open para empezar el desensamblado del archivo. W32Dasm ahora desensamblará calc.exe. Note el tiempo que toma para desensamblar en comparación con el tiempo que tarda en el ejercicio 3.

Nota: Si el texto desensamblado aparece en la pantalla como "Caracteres Basura", usted necesita seleccionar y salvar una fuente por defecto que trabaje en su sistema. Para hacer esto, seleccione **Font** del menú **Disassembler**. Un segundo menú aparecerá que tiene como opciones **Select Font** y **Save Default Font**. Use **Select Font** para seleccionar una fuente que usted crea que trabaje mejor en su sistema. Cuando elija la fuente apropiada, marque **Save Default Font** para hacerla fuente por defecto.

Guardando el texto desensamblado y creando un Project File

En este ejercicio usted guardará el texto desensamblado[2] en un archivo ASCII ".alf" el cual podrá ser usado para recuperar rápidamente el listado de código sin hacer el proceso de desensamblado (a esto se le denomina Project File). Los archivos Project son especialmente útiles cuando usted analiza repetidamente archivos largos que tardan mucho tiempo en desensamblarse.

2.1 Seleccione "**Save Dissassembly Text File and Create Project File**" del menú de disassembler o presione el botón  de la barra de herramientas.

2.2 Aparecerá la ventana "Guardar Archivo" con el nombre por defecto para salvar el archivo desensamblado con una extensión *.alf. El directorio por defecto para salvar estos archivos es DRIVE:\W32DASM DIRECTORY NAME\WPJFILES, pero puede cambiarse.

Es recomendable que use la extensión y el directorio por defecto. Cuando el archivo *.alf es creado, un archivo *.wpj se crea automáticamente. Estos dos archivos siempre deben estar en el mismo directorio para funcionar correctamente. Los archivos *.alf y *.wpj no deben ser modificados por el usuario. Si usted quiere editar el texto desensamblado que reside en el archivo *.alf, haga una copia y renombre el archivo.

Presione aceptar para salvar el archivo desensamblado y el Project file.

Abriendo un archivo Project existente

En este ejercicio usted abrirá el archivo Project "wpj" que fue salvado en el ejercicio 2.

3.1 Seleccione "**Open Project File**" del menu **Project**. El

archivo calc.wpj aparecerá en la ventana de selección. Seleccione este archivo y presione aceptar.

3.2 El archivo desensamblado de calc.exe se abrió. Note que el tiempo en que se cargo el Project File fue mucho menor que el tiempo que tardó en desensamblarse en el ejercicio 1.0.

Navegando el texto Desensamblado

En este ejercicio usted aprenderá como navegar a través del texto desensamblado usando las funciones de Search y Goto del W32Dasm.

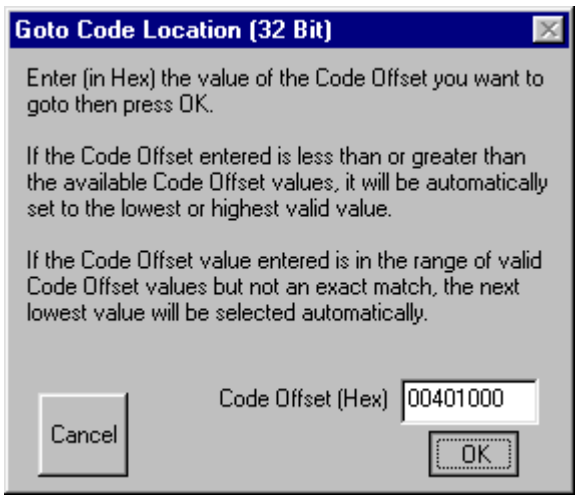
4.1 **Goto Code Start:** Presionando el botón  de la barra de herramientas (cuarto desde la izquierda) o seleccionando "**Goto Code Start**" del menú **Goto** ó presionando Ctrl + S, el listado desensamblado estará al principio del código (Code Start). La parte sombreada con azul es donde el listado es enfocado. El usuario podrá mover la parte sombreada haciendo doble click en cualquier línea de texto o presionando las teclas arriba y abajo.

Note que el listado de código del principio (Code Start) no es necesariamente donde el programa empieza la ejecución. El punto de entrada de la ejecución (o sea la del programa) se llama **Program Entry Point**.

4.2 **Goto Program Entry Point:** Presionando el botón  de la barra de herramientas (quinto desde la izquierda) ó seleccionando "**Goto Program Entry Point**" del menú de **Goto** ó presionando F10, el texto desensamblado se situará en el Program Entry Point. Este es el lugar donde el programa empieza su ejecución. Este es el sitio donde el debugger automáticamente parara cuando sea cargado con algún programa.

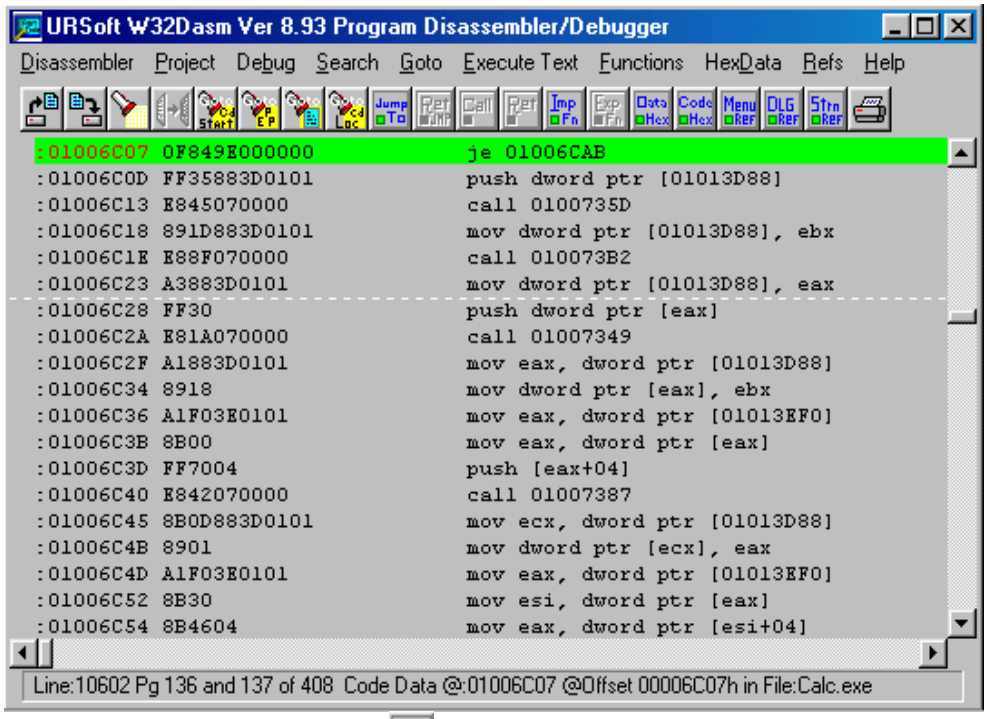
4.3 **Goto Page:** Presionando el botón  de la barra de herramientas (sexto desde la izquierda) ó seleccionando "**Goto Page**" del menú de **Goto** ó presionando F11, aparecerá una caja de dialogo donde teclear el numero de pagina a la que se desea ir en el listado desensamblado.

4.4 **Goto Code Location:** Presionando el botón  de la barra de herramientas (séptimo desde la izquierda) ó seleccionando "**Goto Code Location**" del menú de **Goto** ó presionando F12, aparecerá una caja de dialogo donde teclear la dirección a la que se desea ir en el listado desensamblado.

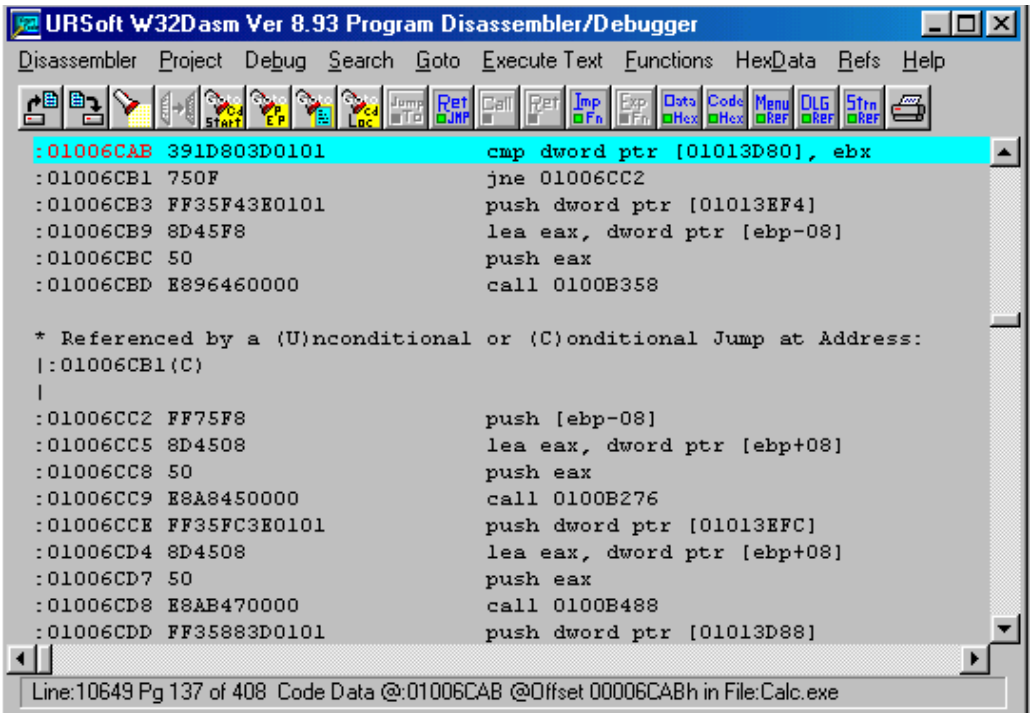


La dirección debe ponerse en formato hexadecimal. Con el archivo calc.exe cargado en el W32Dasm, cambie la dirección que desea ir a 01006C07 y presione ok. El listado ahora estará en la dirección 01006C07 que es una instrucción JE 01006CAB. Fijese que el sombreado se torno verde. El color verde indica que la instrucción JE es una instrucción JUMP valida para ser ejecutada (Texto JUMP).

4.5 **Ejecutando Text Jump:** La Función **Text Jump** es solo activada cuando una instrucción jump valida esta sombreada. La parte sombreada se tornara verde y el botón **Jump To** de la barra de herramientas estará activo.



Presionando el botón  de la barra de herramientas (octavo desde la izquierda) ó seleccionando "**Execute Jump**" del menú **Execute Text** ó presionando la flecha derecha del teclado, el listado desensamblado *irá* a la dirección indicada por la instrucción. En este ejemplo irá a la dirección:01006CAB que es una instrucción CMP DWORD PTR [01013D80], EBX.



Para regresar al sitio original vea el ejercicio 4.6

4.6 **Regresar Del Ultimo Salto:** La función **Return From Last Jump** es solo activada cuando se ha hecho un Jump (Flecha Derecha). El botón 

de la barra de herramientas se activara cuando esta condición exista. Presionando el botón 

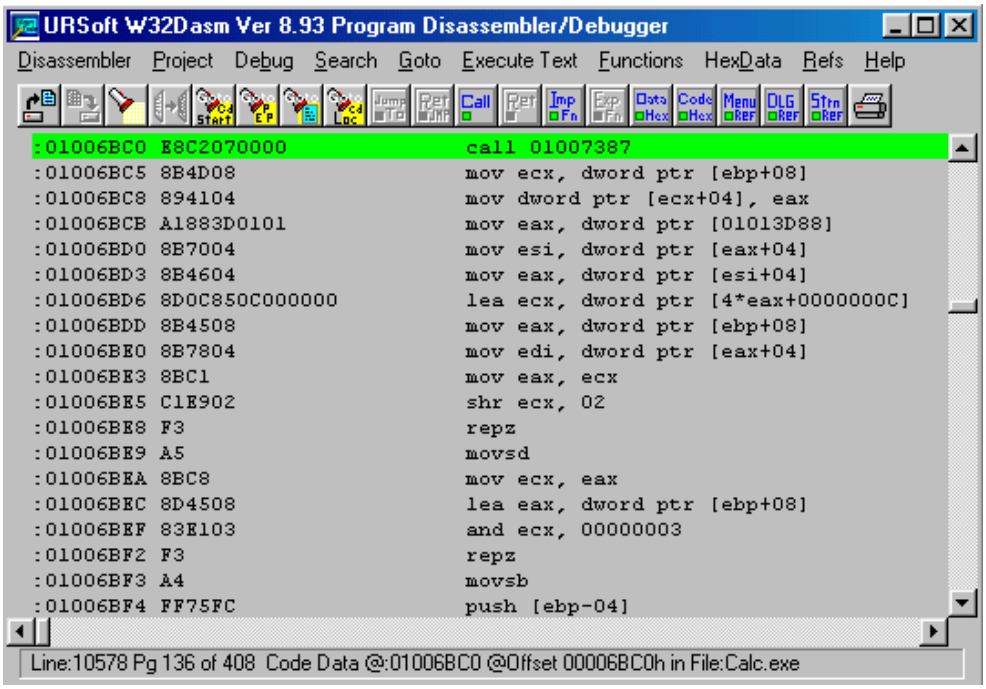
(noveno desde la izquierda) de la barra de herramientas ó seleccionando **"Return From Last Jump"** del menú **Execute Text** ó presionando Ctrl + Flecha Izquierda, el listado *regresará* a la dirección del ultimo salto ejecutado.

4.7 **Ejecutar un Text Call:** La función **Execute Text Call** es activada cuando una instrucción Call valida esta sombreada. La parte sombreada se tornará verde y el botón 

de la barra de herramientas se activará cuando esta condición exista. La parte sombreada ahora estará verde y el botón de la barra de herramientas se activara

Presionando el botón 

de la barra de herramientas (décimo desde la izquierda) ó seleccionando **"Execute Call"** del menú de Execute Text ó presionando la Flecha **derecha** del teclado, el texto desensamblado ahora ira a la localización especificada por la instrucción call. En este ejemplo el listado ira a la dirección

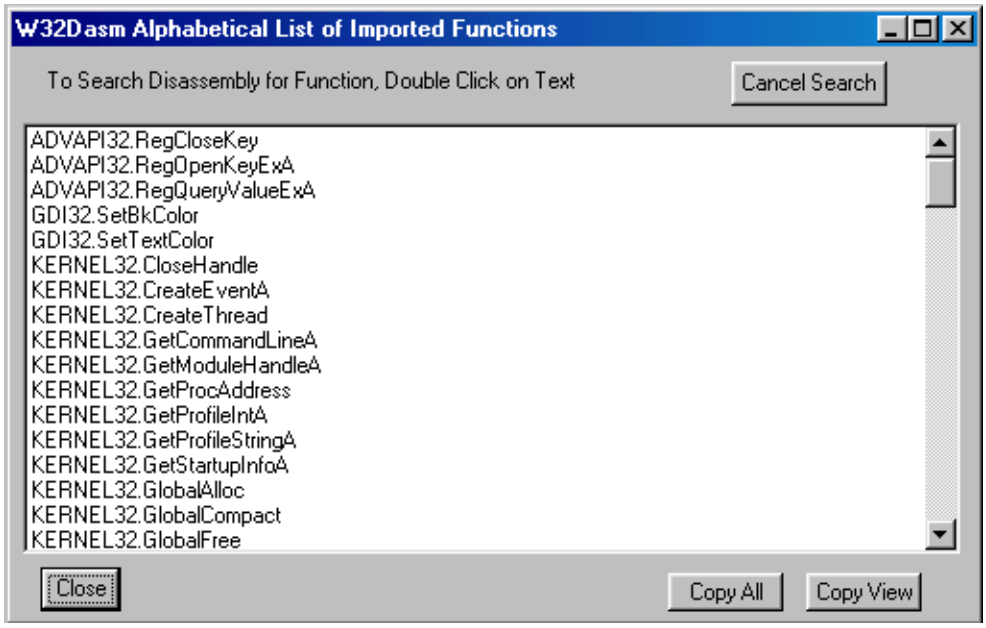


01007387 que es una instrucción MOV EAX, DWORD

PTR [ESP+04]. Para retornar a la dirección original mirar el ejercicio 4.8.

4.8 **Retornar Desde la Ultima Call:** La función **Return From Last Call** es solo activada cuando una Execute Call se ha hecho. (Ver ejercicio 4.7). El botón  de la barra de herramientas se activara cuando esta condición exista. Presionando el botón  de la barra de herramientas (decimoprimero desde la izquierda) ó seleccionando **"Return From Call"** del menú de **Execute Text** ó presionando la Flecha **Izquierda** del teclado, el texto desensamblado retornara a la dirección de la ultima Call ejecutada.

4.9 **Encontrando Funciones Importadas:** Para buscar la lista de las funciones importadas (imported functions) presione el botón  de la barra de herramientas (doceavo desde la izquierda) ó seleccionando **"Imports"** del menú de Functions. Aparecerá una caja de dialogo:



Se mostrarán todas las funciones importadas (import function references) encontradas en el código por el W32Dasm. Haciendo doble click en cualquiera de estos items, el texto desensamblado ira a la localización de la función.

NOTA: Puede haber mas de una referencia para la función en el texto desensamblado. Haciendo doble click

repetidamente en la misma función irá a todas las referencias de esa función. En este ejemplo, si hacemos doble click en el primer ítem Advapi32.RegCloseKey, el texto desensamblado ira a la dirección 01007260 que es una instrucción CALL DWORD PTR [01001008]
La caja de dialogo de las funciones importadas tiene las opciones **Copy View** y **Copy All**, si usted desea copiar el texto visible o el contenido entero de la caja de dialogo a un editor de texto es su opción. Después de presionar el botón de copiar, vaya a su editor de texto y presione pegar para copiar el texto.

4.10 Encontrando funciones exportadas: Para buscar las funciones exportadas (exported functions) presione el botón  de la barra de herramientas (treceavo de la barra de herramientas) ó seleccionando **"Exports"** del menú de functions. Una caja de diálogo aparecerá con toda la lista de las funciones importadas encontradas en el código por el W32Dasm. Haciendo doble click en cualquiera de esos items, el texto desensamblado aparecerá en la localización de esa función.
NOTA: El archivo Calc.exe no tiene funciones exportadas, por eso el botón no esta activado. Trate de desensamblar un archivo .dll que siempre tiene funciones exportadas.
La caja de dialogo de las funciones exportadas tiene las opciones **Copy View** y **Copy All**, si usted desea copiar el texto visible o el contenido entero de la caja de dialogo a un editor de texto es su opción. Después de presionar el botón de copiar, vaya a su editor de texto y presione pegar para copiar el texto.

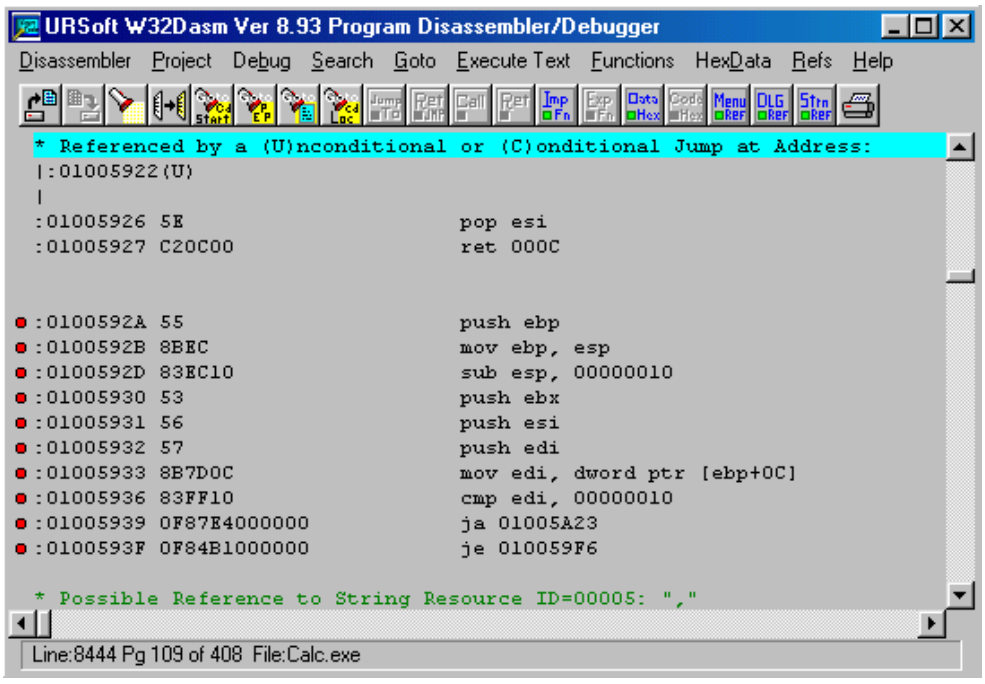
4.11 Encontrando Otras Referencias: Para buscar la lista de referencias para Menú  , Dialog  ó String Data  presione el botón correspondiente de la barra de herramientas ó seleccione el ítem correspondiente del menú de Reference. Una caja de dialogo aparecerá con toda la lista de las funciones encontradas en el código por el W32Dasm. Haciendo doble click en cualquiera de estos items, el texto desensamblado aparecerá en la localización de esa función. Cada caja de dialogo del menú de References tiene las opciones **Copy View** y **Copy All**, si usted desea copiar el texto visible o el contenido entero de la caja de dialogo a un editor de texto es su opción. Después de presionar el botón

de copiar, vaya a su editor de texto y presione pegar para copiar el texto.

Imprimiendo/Copiando el texto Desensamblado

5.1 **Imprimiendo:** Presionando el botón  de la barra de herramientas (el ultimo de la barra de herramientas) ó seleccionando **"Print"** del menú **Disassembler**, una ventana de impresión aparecerá preguntando que páginas desea imprimir, las seleccionadas o todas las paginas del texto desensamblado.

5.2 **Imprimiendo/Copiando Lineas Seleccionadas Del Texto:** W32Dasm tiene las opciones de imprimir o copiar líneas individuales del texto desensamblado. Para seleccionar las líneas, haga click en el margen izquierdo de la línea del texto que desea imprimir/copiar. Un pequeño punto rojo aparecerá en la margen izquierda. Para selecciones múltiples líneas, haga click de nuevo en otra línea que desea imprimir/copiar mientras presiona la tecla **Shift**. Una serie de puntos rojos indican el rango de las líneas seleccionadas.



Para Imprimir: Seleccione **Print**  en la barra de herramientas ó del menú de disassembler. Presione ok para imprimir.

Para Copiar: Seleccione **Copiar Líneas Del Texto**  en la barra de herramientas ó del menú de disassembler. Usted ahora puede usar la función de pegar en cualquier editor de texto.

Cargando una Aplicación de 32 bits en el Debugger

En este ejercicio usted cargara e iniciara calc.exe en el debugger.

6.1 Desensamblar el programa Calc.exe.

6.2 Seleccione **"Load Process"** del menú **Debug** ó presione Ctrl + L. Aparecerá una caja de dialogo que tiene la opción de introducir un comando adicional en el programa antes de ser cargado. Por ahora solo presione LOAD. Calc.exe ahora esta cargado en el debugger y la ventana principal del W32Dasm se movió y adoptó otro tamaño mientras dos nuevas ventanas de debugging (Ver figuras 1 y 2) se crearon. Estas ventanas son llamadas Ventana inferior izquierda y Ventana inferior derecha. Después de inicializar el programa calc.exe WDasm automáticamente se situará en el Program Entry Point. Para el archivo calc.exe se situará en la dirección:010119E0. La Ventana inferior izquierda tiene varias cajas que contienen los Registros, los Flags, BreakPoints, Active DLLs, Code Segment Registers, Trace History, Event History, User Set Memory Addresses, y Data Displays. El Data Display 1 tiene cuatro formatos (Dword, Word, byte, e Instruction) de los cuales usted podrá elegir el que mas le guste. El Source of Data es seleccionado por los botones de la izquierda. El Data Display 2 muestra los cuatro formatos de datos en un solo panel y además tiene un botón para seleccionar la dirección de la Data Display 1. También hay disponibles dos cajas para introducir direcciones de memoria a las cuales quiere acceder. Esta ventana tiene más botones con funciones de debugging que

se explicaran mas adelante en este tutorial.

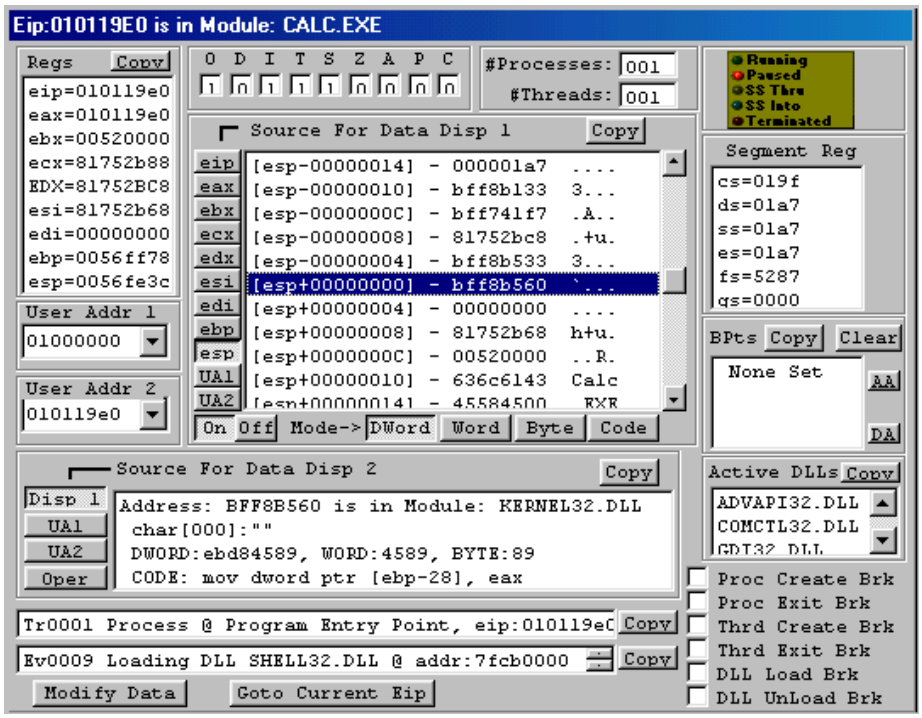


Fig 1. Lower Left Hand Debugger Window

La Ventana inferior izquierda tiene una ventana secundaria con varios botones para debugging que serán explicados mas adelante.

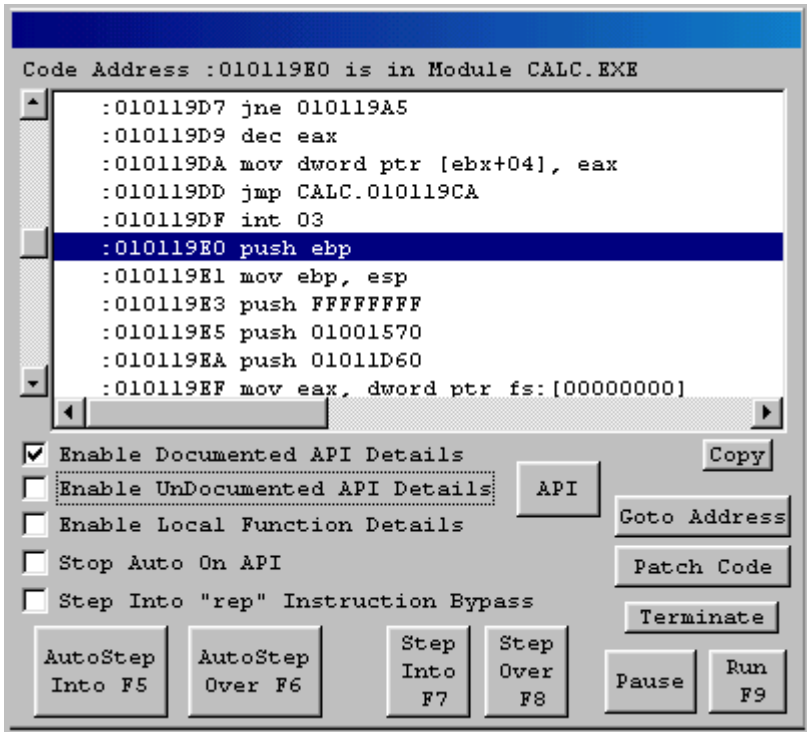


Fig 2. Lower Right Hand Debugger Window.

Usted ahora esta listo para usar el debugger.

Corriendo, pausando y terminando un programa

En este ejercicio usted correrá calc.exe desde el debugger, lo pausara, y luego lo terminara.

7.1 Para correr calc.exe desde el debugger, presione el botón **Run** en la Ventana inferior derecha ó presione **F9**. El programa calc.exe aparecerá normalmente. La Memoria, los Registros y Flags son mostrados en la Ventana inferior izquierda con varias opciones de mostrar los datos. La información de los Active DLLs, BreakPoints, Trace History, Event History y el Estatus del debugger son mostrados en esta ventana.

Para pausar el programa calc.exe, presione el botón **Pause** en la Ventana inferior derecha ó presione la **Barra espaciadora**. Esta acción pausará la ejecución del programa donde sea que el programa muestre algún mensaje al tiempo que usted presione pause. El lugar exacto donde el programa se detiene depende mucho del programa y de la función que estaba llevando a cabo cuando usted hizo pause. Una vez el programa es pausado usted puede usar la opción de seguir paso a paso la ejecución del programa (single stepping).

Para terminar el programa calc.exe usted puede usar el procedimiento normal para cerrar un programa[3] ó puede presionar el botón **Terminate** en la Ventana inferior derecha o Ctrl.-T. Cuando se usa la terminación normal las ventanas del debugger permacen abiertas y registran todos los eventos pertinentes a la terminación del programa. Cuando usa Terminate usted recibirá un mensaje de advertencia sobre la terminación el cual puede cancelar. Si elige terminar el programa terminará y las ventanas del debugger se cerraran automáticamente.

Single Stepping

En este ejercicio usted usará las cuatro funciones del single stepping en el debugger[4].

8.1 Cargue de nuevo el programa calc.exe en el debugger.

8.2 Después de que el programa cargue y pare en el Program Entry Point, usted puede ejecutar cada instrucción del programa usando los botones **Single Step Into** ó **Single Step Over** en la Ventana inferior derecha. Las teclas **F7** y **F8** también pueden ser usadas. La diferencia entre Single Step Into y Single Step Over es que el Single Step Into ejecutara cada instrucción tal como aparece mientras el Single Step Over "saltara" las funciones repetidas como REP MOVSB, y también saltara las instrucciones Call (no entrara dentro de ellas).

8.3 Mientras usted tracea paso a paso a través del programa, puede observar los cambios en los registros y en la memoria cuando cada instrucción es ejecutada.

8.4 Los botones **Auto Single Step Into (F5)** y **Auto Single Step Over (F6)** harán su respectiva función automáticamente. Para pararlos usted puede presionar los botones **Pause, Single Step Into, Single Step Over** ó **Run**.

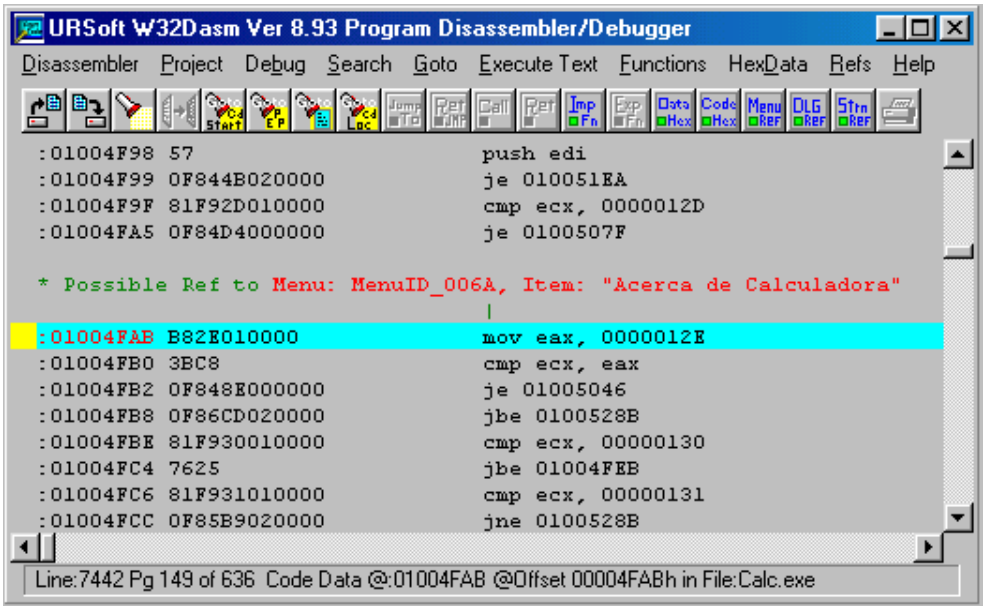
Colocando y activando BreakPoints a un programa

En este ejercicio usted aprenderá como colocar un breakpoint y como activarlo y desactivarlo.

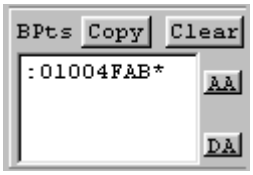
9.1 Cargue el programa calc.exe en el debugger.

9.2 Usando la función Goto Code Location coloque la dirección :01004FAB. Esta localización tiene como instrucción MOV EAX, 0000012E. Usted pudo haber obtenido esta instrucción usando la función Menú Reference y haciendo doble click en Menuid_006a, ítem: "acerca de calculadora". Esta instrucción es parte de la rutina que determina la

función seleccionada del menú **VIEW** del programa calc.exe. Mientras esta instrucción esté sombreada usted puede colocar un BreakPoint presionando la tecla **F2** ó mientras se presiona la tecla **Ctrl** hacer click izquierdo en la margen izquierda (como cuando vamos a copiar pero con Ctrl). Si se usa el mouse no requiere que la instrucción este sombreada. Cuando se coloca un BreakPoint la margen izquierda se tornara amarilla, indicando un BreakPoint.



Si la instrucción del BreakPoint no es la que esta sombreada, la línea entera se tornara amarilla. Note que la dirección del BreakPoint se muestra en la Ventana inferior izquierda. Esta tiene el símbolo ***** al lado de ella indicando que es un BreakPoint activo.

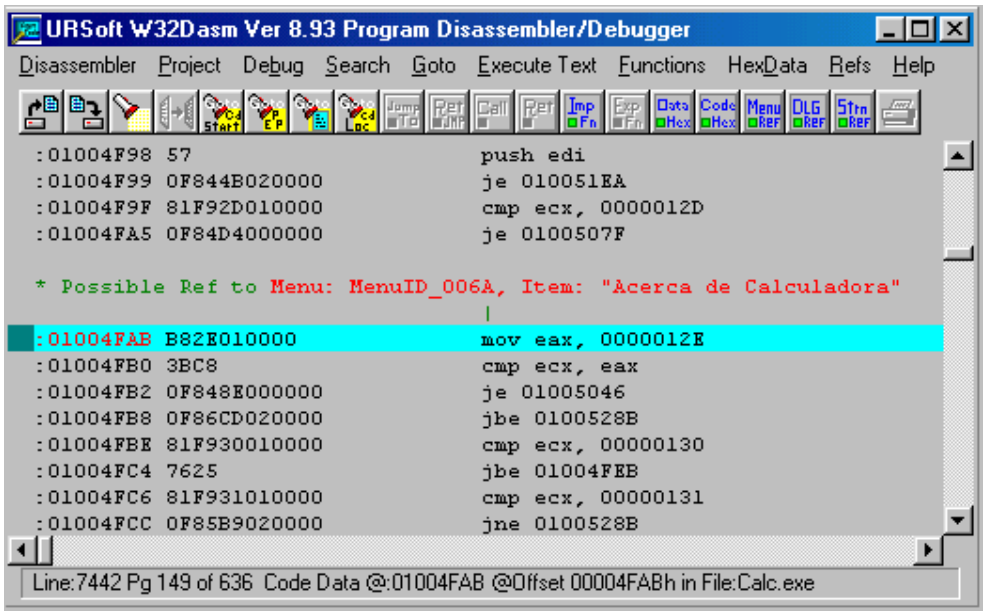


Presionando la tecla **F2** mientras la instrucción donde está el BreakPoint esta sombreada quitará el breakpoint. También se puede hacer con el Mouse como explicamos anteriormente. Para el siguiente paso, dejar el breakpoint en la dirección :01004FAB. Vuelva al texto desensamblado donde está la instrucción actual (en este caso en el program entry point), presionando el botón **Goto Current Eip** que

esta en la Ventana inferior izquierda. Este botón es un camino fácil para ir a la instrucción actual después de haber navegado el texto desensamblado.

9.3 Ahora que el BreakPoint está puesto, presione el botón **Run (F9)** para iniciar el programa calc.exe. Ahora seleccione **"acerca de calculadora"** en el menú de **ayuda**. El debugger ahora parará la ejecución del programa (break) en la dirección :01004FAB.

9.4 Un Breakpoint puede desactivarse. Para hacer esto seleccione la dirección del breakpoint en la caja de los breakpoints y haga click **Derecho**. Cuando un breakpoint es desactivado, el símbolo ***** desaparecerá de la dirección del BreakPoint y en el texto desensamblado pasara de amarillo a verde oscuro.

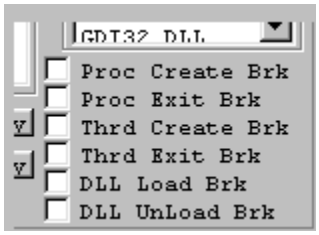


El BreakPoint puede ser reactivado repitiendo la operación anterior. Si usted necesita desactivar todos los breakpoint que haya puesto de una sola vez puede presionar el botón **DA** que esta a la derecha de la lista de BreakPoints. Presione el botón **AA** para activar todos los BreakPoints. Si usted desea quitar (borrar) todos los BreakPoints, presione el botón **Clear**. Un mensaje de advertencia aparecerá para confirmar la eliminación de los BreakPoints.

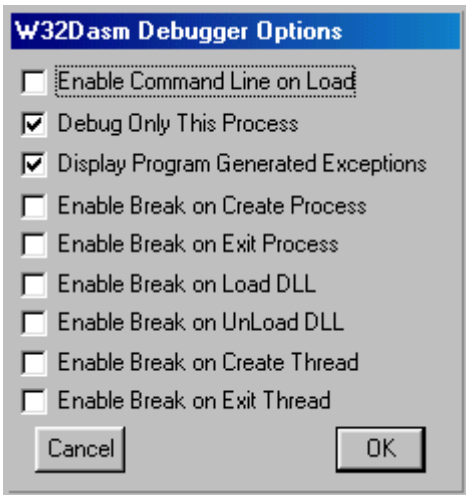
9.5 **Goto BreakPoint Location:** Usted puede ir a la localización de un BreakPoint rápidamente haciendo **Doble Click Izquierdo** en el BreakPoint al que desea ir. El BreakPoint

puede estar activado ó desactivado para esta operación.

9.6 **Automatic Breaks On Event:** Usted puede tener el debugger configurado para parar la ejecución del programa cuando sucedan ciertos eventos como cuando un **DLL es cargado ó cerrado**, cuando una **línea es creada ó cerrada**, ó cuando un **Proceso es creado o cerrado**. Esto es configurable en las opciones que tiene la Ventana inferior izquierda.



Estas cajas para selección pueden ser configuradas por el usuario como opciones por defecto seleccionando lo que se quiere en el **Debugger Options** del menú de Debug.

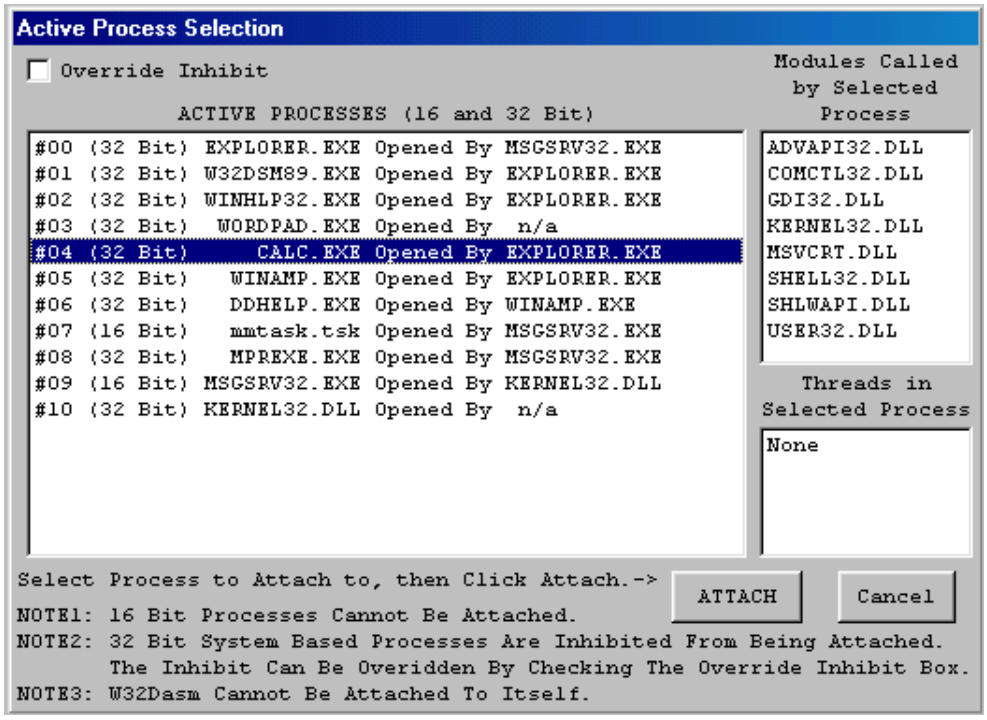


Adjuntando a un proceso activo (Active Process)

En este ejercicio usted aprenderá como adjuntar al debugger un proceso que esta corriendo actualmente en windows.

10.1 Empiece el programa calc.exe en windows.

10.2 Seleccione **"Attach To An Active Process"** del menú de Debug . Una caja de dialogo aparecerá



Aparecerá una lista con todos los procesos activos que se están corriendo en ese momento en Windows. Esta lista contiene referencias de ambos programas de 16 bit y 32 bit. Solo los programas de 32 bit pueden ser adjuntados. Pero, algunos programas de 32 bis estan deshabilitados para adjuntar porque son archivos importantes de Windows y si se adjuntan pueden causar que el computador se bloquee. De todas formas si usted quiere adjuntar estos programas, hay una opción que las habilita que esta en la parte superior izquierda **"Override Inhibit".** Si usted activa esta opción podrá adjuntar los programas que antes no podía. Ver la ADVENTENCIA de abajo. La ventana también muestra que módulos (DLLs) son llamados y que Threads son creados por el programa que selecciono en la lista de programas.

10.3 Busque calc.exe en la lista y selecciónelo haciendo click sobre él.

10.4 Presione el botón **"Attach"**. El programa calc.exe ahora será cargado en el debugger. La principal diferencia entre adjuntar y cargar un programa en el debugger es que

adjuntándolo el debugger no parara en el Program Entry Point porque el código del programa ha sido ejecutado antes. Usted puede romper el programa presionando cualquier botón de **Single Step** ó **Pause**.
ADVERTENCIA: Cuando un programa es adjuntado, este terminara cuando el W32Dasm es cerrado. Cuando se adjunta un archivo importante de Windows, asegúrese de no pausarlo ó terminarlo porque esto podría provocar que windows no operara correctamente. Si esto pasa reinicie el computador.

Modificar registros, flags, memoria e instrucciones

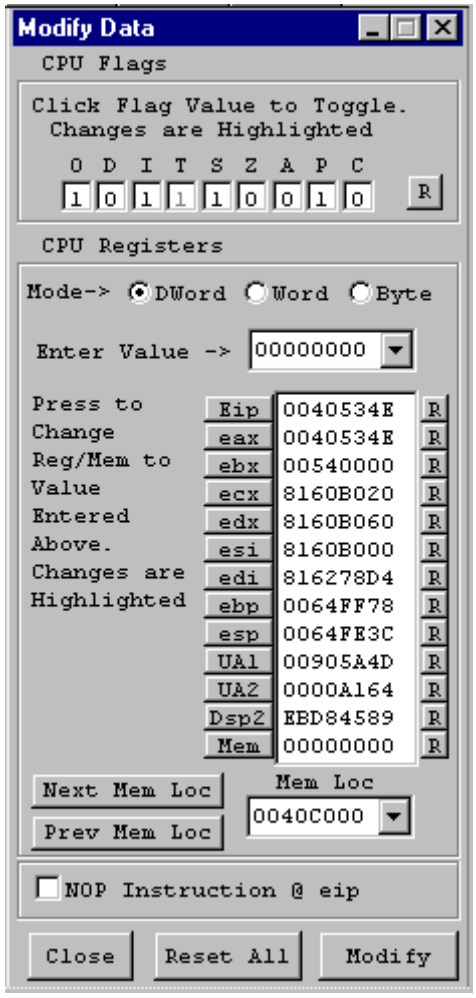
En este ejercicio usted aprenderá como modificar la registros, los flags, las instrucciones y la memoria con el debugger.

11.1 Cargue el programa calc.exe en el debugger. No inicie el programa (F9).

11.2 Presione el botón **Modify Data** que esta en la Lower Left Hand Debugger Window. Una caja de dialogo aparecerá

11.3 **Modificar Flags:** Para cambiar el valor de un Flag, hacer click izquierdo en el flag que se desea modificar. Este cambiara de 0 a 1 en cada click. Cuando un valor de un flag no es el mismo que el valor original, este estará sombreado en azul. Ninguna modificación se ara mientras no presione el botón **"Modify"**. **Restaurando El Valor Del Flag:** En cualquier momento antes de presionar el botón **Modify**, usted puede Restaurar (RESET) todos los Flags al valor original presionando el botón **R** en la caja de los Flags.
NOTA: El Flag T no puede ser modificado

11.4



.

11.5 **Modificando Los Valores De Los Registros:** Para cambiar un valor de un registro, teclee el valor deseado en la caja de **"Enter Value"**. Dependiendo del modo que está seleccionado en la caja de **"Mode"**, usted tiene la opción de cambiar el registro **DWORD** (ej: eax) ó el registro **WORD** (ej: ax) ó el registro BYTE (ej: ah, ó al). Presione el botón designado para el registro que desea modificar. Los registros cambiados aparecerán sombreados en azul. Ninguna modificación se hará mientras no presione el botón **"Modify"**.

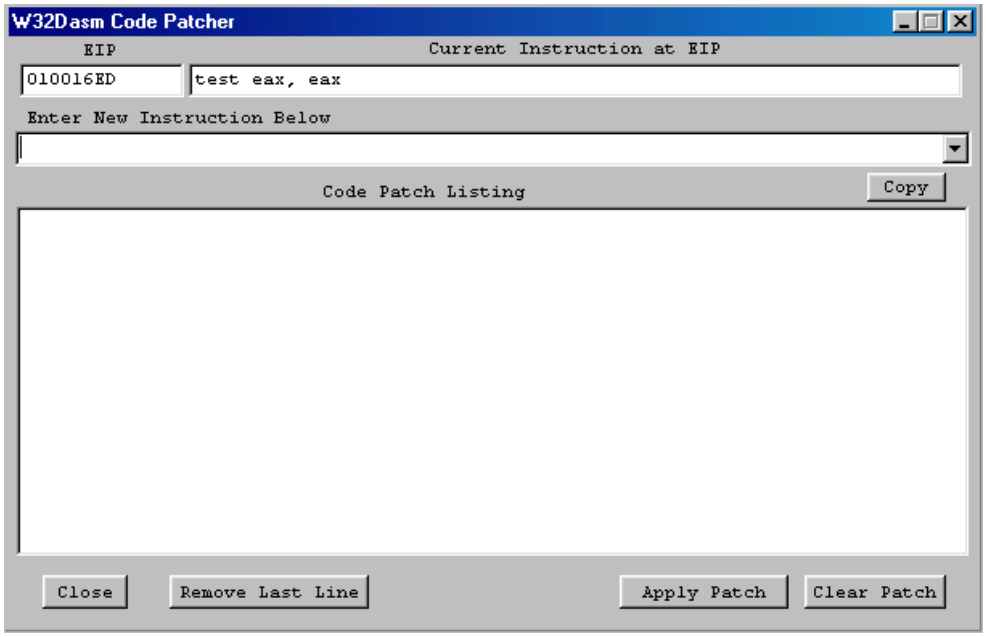
Restaurando Los Valores De Los Registros: Cada registro tiene un botón **R** para restaurarlo al valor original.

11.6 **Modificando Valores En Memoria:** Para cambiar un valor en una localización en memoria (DWORD, WORD, ó BYTE), introduzca el valor deseado en la caja de **"Enter Value"** y teclee la dirección de memoria en la caja de dialogo de **Mem Loc**. Después presione el botón **"Mem"** para

modificarlo. Usted puede ajustar la dirección de memoria presionando los botones Mem Loc ó Prev Mem. El valor en la dirección de memoria puede aumentar ó rebajar por una DWORD, WORD ó BYTE dependiendo del modo seleccionado en la caja de mode. Ninguna modificación se hará mientras no presione el botón **"Modify"**. Las direcciones especificadas en las cajas **User Addr 1** y **User Addr 2** y **Display 2** en la Ventana inferior izquierda pueden ser modificadas presionando el botón apropiado en la caja de dialogo de Modify Data.

Restaurando Los Valores En Memoria: En cualquier momento antes de presionar el botón Modify, usted puede Restaurar (RESET) el valor modificado al valor original, presionando el botón **R** que corresponda a la que se desea restaurar.

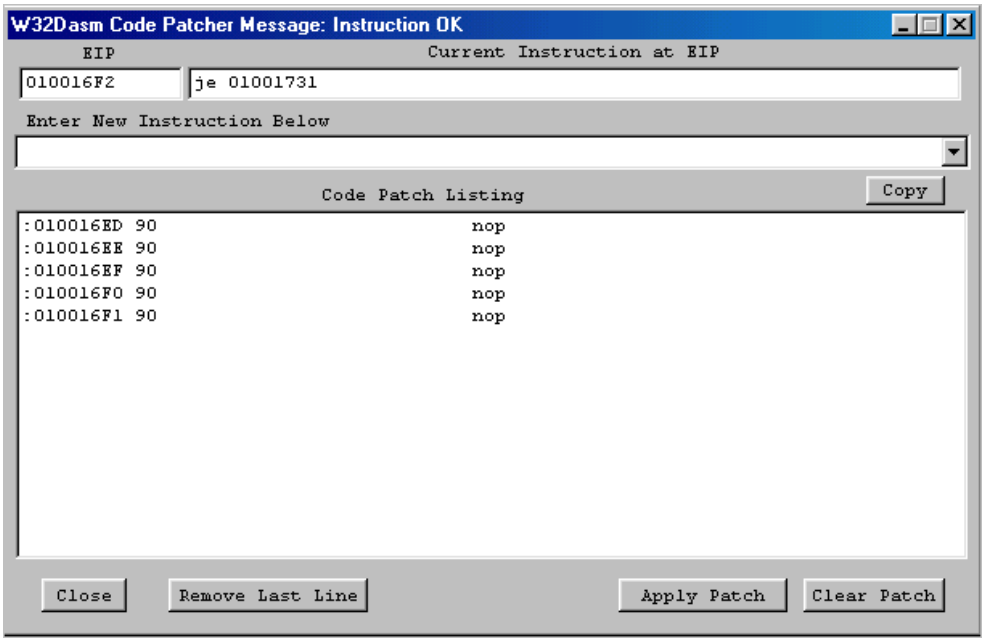
Modificando Instrucciones: Usted puede **NOP**ear cualquier instrucción de la EIP seleccionada mediante la opción **"NOP Instruction @ EIP"** antes de presionar el botón **Modify**. W32Dasm automáticamente determinara cuantas instrucciones NOP debe insertar para nopear la línea entera. Para modificar instrucciones en otra dirección, use el botón **Code Patch** en la Ventana inferior derecha. Esta abrirá una caja de dialogo Code Patcher. La dirección de la instrucción(es) para parchear es determinada por la línea que este sombreada en la Ventana inferior derecha. Usted puede cambiar la dirección: (1.) Usando las opciones de Single Stepping, (2.) Usando el botón de **Goto Address** ó (3.) Usando las opciones de arriba/abajo en la Ventana inferior derecha. Una vez tenga sombreada la instrucción que quiera cambiar, usted puede escribir las instrucciones en la caja de **Enter New Instruction Below**. Cuando usted presione enter, la nueva instrucción aparecerá en la lista de la caja del Code Patcher.



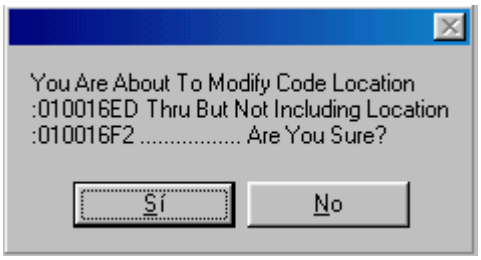
Si la instrucción es invalida ó está escrita incorrectamente no será admitida. Como una guía general, para saber si una instrucción es válida, mire la ventana principal del texto desensamblado, por ejemplo. Algunas instrucciones requieren que un tamaño (ej: "dword ptr" ó "byte ptr") sea usado. Todos los valores numéricos deben estar en notación hexadecimal. Para borrar la lista entera del Code Patch, presione el botón **Clear Patch**. Para borrar solo la ultima línea en la lista del Code Patch, presione el botón **Remove Last Line**. Cuando se modifique la instrucción usted necesita estar seguro de que la instrucción a ser reemplazada cubra el mismo numero de bytes que la nueva instrucción. Use la instrucción NOP para cubrir los bytes que no son usadas por la nueva instrucción. Cuando la nueva instrucción esta lista, usted puede ahora modificar el programa presionando el botón **Apply Patch**. Aparecerá una caja de dialogo confirmando la acción. Si usted escoge "SI" el programa sera modificado.

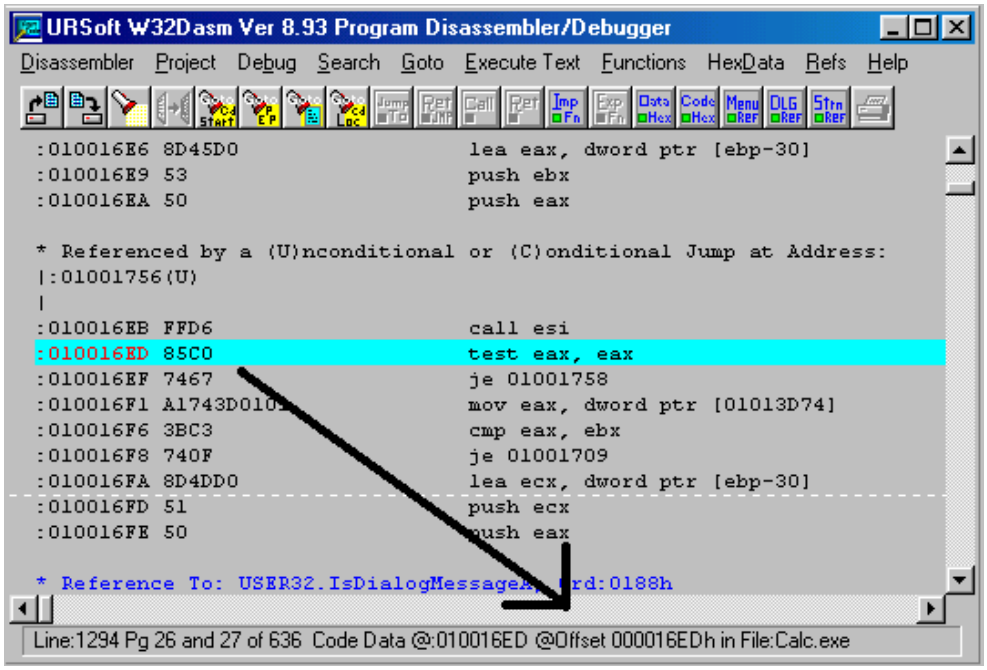
NOTA 1: Algunas modificaciones no se pueden hacer porque algunas áreas en memoria están protegidas contra escritura, como es el caso del área de memoria del Kernel32.dll.

NOTA 2: Las modificaciones solo son mostradas en la Ventana inferior derecha. La ventana principal del W32Dasm solo mostrará los valores originales.



NOTA 3: las instrucciones modificadas son solo temporales. El ejecutable original ó el archivo DLL no son modificados. Para hacer los cambios permanentes use un editor Hexadecimal y reemplaze en la localización adecuada. La localización de la línea a cambiar puede ser conocida mirando en el status[5] de la ventana principal del W32Dasm mientras la instrucción a cambiar este sombreada. En el status se muestran varios valores pero para encontrar la instrucción en el editor Hexadecimal solo necesitamos el valor del **Offset**.





Explorando los módulos de llamadas (DLLs)

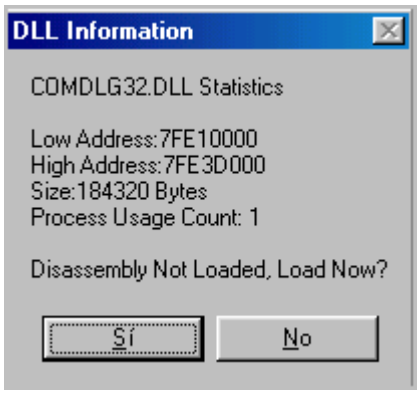
En este ejercicio usted aprenderá como usar el debugger con archivos DLL. Los archivos DLL no pueden ser cargados por ellos mismos en el debugger. Solo archivos ejecutables pueden ser cargados en el debugger. Pero cualquier DLL llamado por el ejecutable puede ser cargado en el debugger después de que el ejecutable es cargado en el debugger.

12.1 Cargue el programa Notepad.exe en el debugger y presione **F9** para iniciar el programa

12.2 Mire en la caja de listas de las Active DLLs en la Ventana inferior izquierda. Esta caja tiene los nombres de todos los DLLs que actualmente son cargados por el programa. Si usted hace doble click en un nombre una caja de dialogo aparecerá dándonos información sobre el DLL. Esta caja de dialogo da la opción de desensamblar el DLL y adjuntarlo en el debugger.

12.3 Presione **"SI"**. Comdlg32.dll ahora esta desensamblado y cargado. Si el archivo Comdlg32.dll ha sido desensamblado y guardado el Project File anteriormente, el archivo Project será cargado. Esto ahorrará un poco de tiempo al

desensamblar el archivo.



12.4 Usted ahora puede usar el debugger en el DLL.
NOTA: A algunos DLLs como el KERNEL32.DLL no se les puede colocar BreakPoints ó modificar sus valores en memoria porque están en una área de memoria PROTEGIDA CONTRA ESCRITURA.

12.5 Presione el botón  de la barra de herramientas y haga doble click en **GetOpenFileNameA**. El texto desensamblado irá a la dirección **:7FE16112** que es una instrucción **PUSH EBP**. Esta instrucción es el principio de la función GetOpenFileNameA en el archivo Comdlg32.dll. Ahora presione **F2** para colocar un BreakPoint en esta instrucción.

12.6 Ahora vaya al programa Notepad.exe e intente abrir un archivo. El debugger parara el programa Notepad.exe y rompió en la instrucción **GetOpenFileNameA** en la dirección :7FE16112 en el archivo Comdlg32.dll.

Detalles de las API de Windows

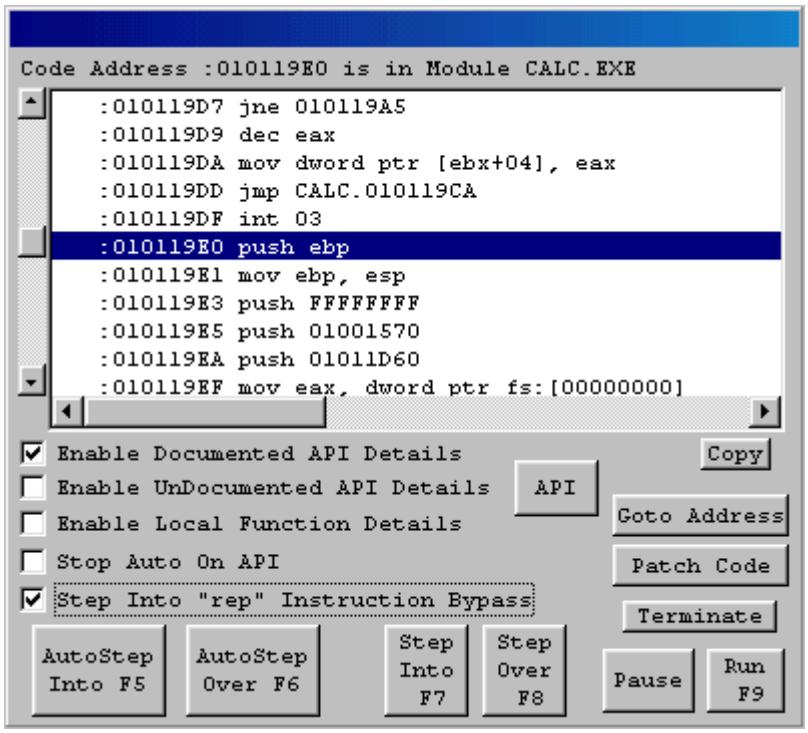
En este ejercicio usted aprenderá como usar las API de windows en el W32Dasm.

13.1 W32Dasm tiene la opción de mostrar al usuario detalles de muchas de las funciones API de Windows. Todas los programas usan generalmente funciones desde el KERNELl32, ADVAPI32, SHELL32, COMCTL32,

COMDLG32, USER32, y GDI32. Pero hay opciones para mostrar **Undocumented Api Details** (detalles indocumentados de las api) y el proceso **Local Function Details** (detalles de funciones locales) con un formato genérico de datos. Los datos son mostrados en Hexadecimal y en Strings. Cargue el programa calc.exe en el debugger pero NO lo inicie (F9). Asegúrese de que la opción **Enable Api Details** esta activada en la Ventana inferior derecha. (esta opción está activada por defecto). De todas formas active la opción **Stop Auto On Api**. Si esta opción no está activada la función de Auto Single Step continuará sin parar cuando una función API es llamada.

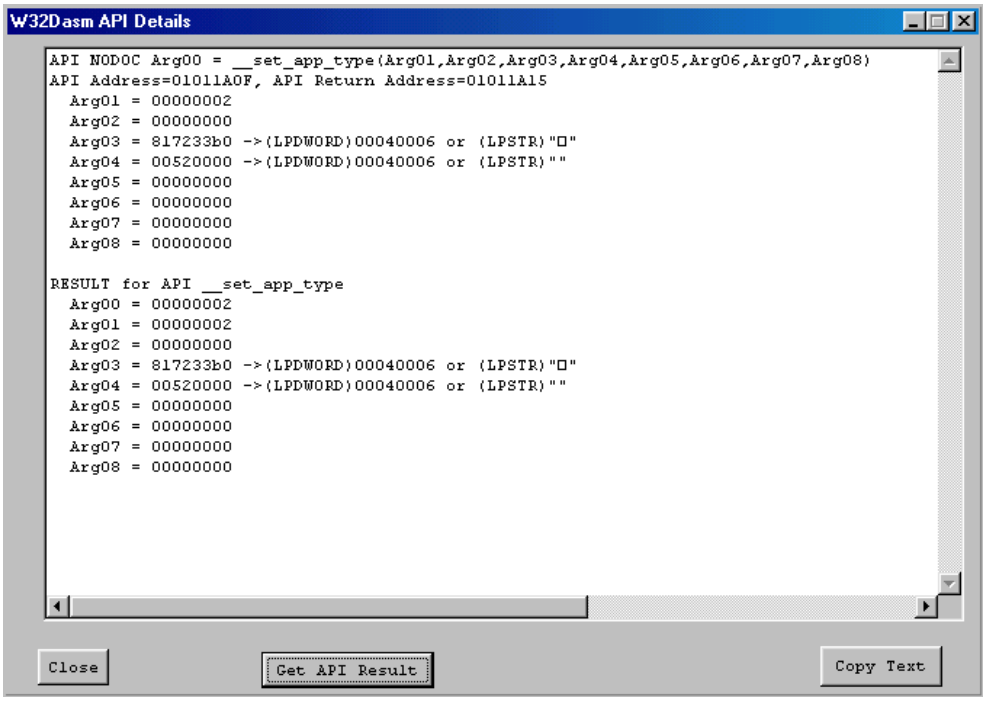
NOTA: Para activar las opciones de **Undocumented Api** ó **Local Function Details**, active las cajas correspondientes.

13.2 Ahora presione **F5** para activar la función Auto Single Step Into. **(también puede usar el single step manual presionando F7)**



13.3 El debugger ahora estará en Auto Single Step Into hasta que se encuentre con una función API que es **__set_app_type** en la dirección **:01011A0F**. Una caja de dialogo aparecerá que muestra la API con el tipo de información y los valores ingresados por la función API. Si se pulsa el botón **Get Api**

Result en esta caja de dialogo, la función API se iniciará y retornara los resultados de la API en la caja de dialogo.



Usted puede usar el botón **Copy Text** para pegar el texto en un editor de texto.

13.4 Presionando de nuevo **F5** ó manualmente la función Single Stepping hará que la caja de dialogo de los detalles de la API se minimice hasta que se encuentre con la siguiente API. La función **API Details** puede ser desactivada en la Ventana inferior derecha.

---

[1] La versión que sigue este tutorial es la de Windows 95.

[2] En muchos sitios lo veréis nombrar como "Listado Muerto".

[3] Eligiendo la opción Exit del programa si la tiene, p.ej, o pulsando sobre el aspa en la esquina superior derecha de la ventan principal.

[4] Hasta aquí hemos visto las funciones de "listado muerto" del Wdasm. Además incluye utilidades de "traceo" o depuración propiamente dichas, poco usadas gracias al "Sice".

[5] Status es la última línea del Wdasm, donde se muestra el offset de la línea de código sombreada.

Capítulo

5

The Tools of the Trade II: Introducción a SoftIce.

Mammon_ y YöÐ@KëR

Uno de los crackers más famosos que aún perduran en la red es Mammon_, autor de algunos de los tutoriales más leídos. Presentamos en este capitulo una rigurosa traducción al castellano de su texto "Introduction to using SoftICE", en el que explica muy resumidamente buena parte de los manuales de Numega "Using SoftIce" y "Command Reference"[i]. A modo de aproximación rápida para poder empezar a usar el programa hemos situado al principio de este capítulo unas líneas de la página de YöÐ@KëR. No hace falta decir que estamos ante la HERRAMIENTA fundamental de la ingeniería inversa de software y que de nuestro grado de conocimiento de ella dependen buena parte de nuestras posibilidades de éxito

Tank y mR gANDALF

¿Cómo instalo SoftICE?

Una vez que se completó el procedimiento de la instalación de SoftICE, este habrá (con tu previo consentimiento) modificado tu autoexec.bat para que SoftICE se cargue cuando inicies tu PC. Si instalas un Menú de Arranque (recomendado) deberías revisar tu autoexec.bat para asegurarse de que este todo en orden. La línea

C:\PROGRA~1\SOFTICE\WINICE.EXE

es la única requerida en tu autoexec.bat para que SoftICE se cargue junto con Windows, de lo contrario no se cargará. Obviamente los directorios puede cambiar.
Después de reiniciar, si presionas CTRL-D en Windows y SoftICE no aparece, quiere decir que SoftICE no está activo o no se cargo adecuadamente. Chequea tu autoexec.bat. Si SoftICE aparece, puedes cerrar la consola presionando CTRL-D otra vez, o F5. Por lo tanto, CTRL-D es el comando empleado para ver SoftICE.

¿Cómo configuro SoftIce?

Después de que SoftICE ha sido exitosamente instalado, hay pocas formas en las cuales necesitarás o querrás configurarlo. Las opciones por omisión son bastante pobres. Primero que nada, para poder establecer breakpoints en API, necesitas tener los símbolos para los módulos correspondientes cargados en SoftICE. Puedes correr Symbol Loader y cargar los módulos manualmente, esto te salvará de tener que reiniciar, pero solo cargará los símbolos durante la actual sesión de Windows.
Una mejor idea es modificar winice.dat, para que los símbolos usados comúnmente sean cargados por omisión. Con agradecimiento, el staff de Numega ha suministrado ya un juego decente y útil de símbolos para que sean cargados dentro del winice.dat, de cualquier forma solo están mencionados (no se cargan).
Para hacer que estos símbolos se carguen, abre winice.dat en cualquier editor de texto. Desplázate hacia abajo hasta las líneas que empiezan con *';EXP='*, y elimina el ';' del comienzo de cada línea EXP. Si realmente sabes lo que estas haciendo, querrás desmencionar una selección de estos. Si eres un novato, deberías deseleccionar todas las líneas que comienzan con EXP.

(EJEMPLO) *;EXP=c:\windows\system\kernel32.dll <== De aquí, elimina el ';' al comienzo de la línea y así sucesivamente.*

Mientras estás EDITANDO tu archivo winice.dat es recomendable que agregues estas lineas:

EXP=c:\windows\system\msvbvm50.dll
EXP=c:\windows\system\msvbvm60.dll

Probablemente no emplees esto por el momento, pero estas líneas cargan los símbolos para el MS VB Virtual Machine. Lo necesitarás mas adelante, si vas a crackear/depurar aplicaciones Visual Basic.

FAQ primeros pasos con el SoftIce

Cuando corro Symbol Loader, se me notifica "SoftIce is not active". ¿Que pasa?

SoftICE, sorpresivamente, esta inactivo ;)
Tal vez SoftICE no se esta cargando al arranque, porque no ha sido incluido en el autoexec.bat (si es así, véase pregunta FAQ #1), o esta intentando cargarse pero falla, en tal caso chequea paso a paso su secuencia de arranque y busca cualquier error que pueda tener la configuración de SoftICE. Así mismo, si acabas de instalar SoftICE, obviamente necesitas reiniciar antes de que este pueda cargar.

SoftICE aparece cada vez que una aplicación marca un GPF. ¿Como prevengo esto?

Por omisión, cada vez que un GPF [Falla de Protección General ('*General Protection Fault*')] ocurre en Windows, SoftICE lo atrapará y la consola aparecerá, permitiéndote depurar el error. Esto puede ser muy irritante para los novatos si no están interesados en depurar el error.

Puedes desactivar este reconocimiento al escribir ***FAULTS OFF*** en SoftICE, y luego presionando F5 para cerrar la consola. Una mucho mejor idea es añadir ***FAULTS OFF*** en tu línea INIT, en winice.dat, claro esta.
La línea INIT en tu winice.dat contiene una secuencia de comandos ('*scripts*') que SoftICE ejecuta automaticante cuando se carga. Este es el mejor lugar para configurar tu consola de SoftICE, incluyendo el tamaño de las ventanas y sus posiciones.

Aquí hay algunos comandos que puedes añadir a tu línea INIT. Hay muchos más. ¡¡¡Lee el Manual que viene con SoftICE!!! [ii]
(Ejemplo) *INIT="FAULTS OFF; LINES 50;WC 20;WD 20;WL;WR;X;CODE ON;X;"*

LINES XX	Establece la consola SoftICE con una altura XX.
WD XX	Hace visible la ventana de datos, con una altura XX líneas.
WC XX	Hace visible la ventana de código con una altura de XX líneas.
CODE ON|OFF	Establece los codigos opcionales de cada instrucción como visible|no visible
FAULTS ON|OFF	Establece SoftICE a permitirle|no permitirle que atrape y muestre los GPFs
X	Cierra la consola de SoftICE. Tu línea INIT debe terminar con X; para que SoftICE no sea inmediatamente visible cuando arrancas Windows. Observa que X es equivalente a presionar F5 mientras SoftICE esta abierto, y no es lo mismo que CERRAR el programa.

Los comandos de la línea INIT están separados por ';'. Pareciera que algunos comandos solo funcionan cuando la consola de SoftICE es cerrada por primera vez. Si encuentras algunos comandos que no están siendo aplicados, intenta ponerlos *después* de la "X;" en tu línea INIT. Puedes agregar otra "X;" para volver a cerrar SoftICE.

¿Qué es un GPF (General Protection Fault)? ¿En que consiste?

En Windows 3.1, caída de una computadora causada por la invasión, por parte de un programa, del espacio de memoria de otro. Las caídas relacionadas con GPFs son comunes en Windows 3.1 y en general en los sistemas operativos con modo multitareas cooperativo, en el cual los programas deben diseñarse muy cuidadosamente para que puedan coexistir en la memoria de la computadora. Otros sistemas operativos más recientes como Windows 95, ofrecen multitareas por preferencias, modo en el cual el sistema operativo interviene en los conflictos por espacio de memoria; tales sistemas operativos ofrecen una operación significativa más confiable. Vea Protected Mode y Real Mode

¿Dónde encuentro tutoriales sobre el uso y la configuración de SoftICE?

Uno de los mejores sitios con info sobre configuración de SoftICE es +Sandmans Code Reversing For Beginners. Ahí, encontrarás el Centro de Recursos para SoftICE (SoftICE Resource Center), el cual contiene cada uno de los tutoriales que necesitarás para tener un conocimiento funcional acerca de SoftIce
Así mismo, obtengan una copia del Manual de SoftICE. Y cuando lo tengas, asegúrate de LEERLO !

¿Cómo establezco un Menú de Arranque para opcionalmente cargar SoftICE?

Agrega esto al inicio de tu config.sys:

```
inicia -->
MENU]
menuitem=ICE,SoftICE
menuitem=NOICE,No SoftICE - RECOMENDADO*
menudefault=NOICE,20**
[ICE]
[NOICE]
[COMMON]
<-- termina
```

**Puse el RECOMENDADO, porque en muchas ocasiones no somos los únicos que empleamos la máquina y no queremos que nadie estropee nuestro magnífico depurador ¿cierto?*
***Este número puede ser modificado según la necesidad. Indica los segundos que se tienen para escoger el CARGAR o NO SoftICE, pasado este tiempo por omisión NO se cargará SoftICE.*

Después, agrega estas 3 líneas a tu autoexec.bat (recordando que debes borrar cualquier referencia ya existente a winice.exe).

Inicia →
IF %CONFIG%==NOICE GOTO NOICE
C:\PROGRA~1\SOFTIC~1\WINICE.EXE ← Aqui la raiz puede cambiar.
:NOICE
← termina

Ahora, por si no lo sabías, cada vez que inicies tu máquina se te preguntará si quieres CARGAR o NO SoftICE. Si no lo sabias realmente deberías preocuparte más por aprender sobre sistemas operativos y el funcionamiento de las computadoras antes de aprender sobre cracking.

¿Porqué obtengo "No Debug Information Found" cuando cargo un módulo dentro del Symbol Loader?

La información para depurar está incluida dentro de los módulos cuando son compilados, para acciones de futuro desarrollo.
Esto le permite a los depuradores integrados de lenguajes como VC++/Delphi etc, que asocien el código del objeto con el código fuente. Cuando una aplicación es lanzada al mercado, la información para depurar no suele ser incluida con el código del objeto, ya que lo único que le provee a los usuarios normales es un archivo de mayor tamaño. Si puedes encontrar la información para depurar contenida en una aplicación liberada, entonces eso es excelente!, pero no te sorprendas porque no venga incluida[iii]. Eso no significa que puedas o no crackear la aplicación, y no es algo porque preocuparse. Ignora el mensaje de error y prosigue normalmente.

SoftIce a fondo: Mammon_ `s Tales to his Grandson

Configurar el Entorno

Soft-Ice tiene un archivo llamado WINICE.DAT que puede ser modificado para configurar tu depurador para un uso óptimo. Todas las opciones de configuración pueden ser ingresadas desde la línea de comandos de Soft-Ice, por lo que puedes querer experimentar antes de sobrescribir los valores predeterminados del programa. En general, las Teclas de Función pueden dejarse sin modificación, pero puedes querer modificar las ventanas visibles, el tamaño y el color de la pantalla, los function export y la pantalla de código. Los cambios pueden ser agregados a la línea INIT="X;" en el archivo WINICE.DAT tal cual

como si fuera en la línea de comandos, de forma que una cadena de inicialización podría ser INIT="Code ON;WR;lines 50;WC25;X;". Ten en cuenta que las líneas "init=" solo pueden contener un número limitado de caracteres, pero puedes especificar múltiples líneas "init=" (siempre y cuando la última termine en "X;").

- Code ON|OFF : Code ON muestra el byte hexadecimal a continuación del listado desensamblado.
- Color : Color es usado para ajustar los colores de pantalla. Los colores son valores hexadecimales de cuatro bits. (por ejemplo: un valor de 1 a 16, o en hex de 0 a F: 0=negro 1=azul 2=verde 3=cyan 4=rojo 5=magenta 6=marrón 7=gris 8=gris oscuro 9=celeste A=verde claro B=cyan claro C=rosa D=magenta claro E=amarillo F=blanco), y cada ajuste de color es una combinación de dos colores, la letra y el fondo (de forma que 0F sería letras negras sobre fondo blanco). Las áreas de la pantalla que pueden ser ajustadas son normal (color del texto normal), bold (color del texto en negrita o resaltado), reverse (color del texto invertido o en negativo), help (color de la línea de ayuda) y line (color de las líneas horizontales). La sintaxis del comando color es **color** *normal bold reverse help line*, de forma que **color 7 F 71 30 2** sería lo siguiente: texto normal gris sobre negro, texto en negrita blanco sobre negro, texto en negativo azul sobre gris, la línea de ayuda negra sobre cyan y las líneas horizontales verde en blanco.
- Exp : Muestra los export symbols de las DLLs y los EXEs. Este comando es vital para entender un listado desensamblado ya que toma las llamadas de Windows de las llamadas internas del programa así no te pasas horas haciendo trace en Kernel32.dll. Escribir **exp** *nombre del módulo* cargará los símbolos de ese módulo. Unas buenas opciones son kernel32.dll/kernel.exe, user32.dll/user.exe y gdi32.dll/gdi.exe. Exp también puede ser usado para mostrar los módulos cargados actualmente (**exp /**) y para listar los símbolos que coinciden con algún patrón (**exp delete** debería listar todos los símbolos en los módulos cargados que comienzan con Delete). Ten en cuenta que es una buena idea cargar los exports en Winice.dat más que de la línea de comando de Soft-Ice.
- Faults ON|OFF : Faults ON hace que Soft-Ice sea invocado para manejar los Errores de Protección General.
- Lines # : Incrementa o decrementa el número de líneas (o el tamaño de la fuente) de la pantalla.
- WC # : Habilita una ventana de código de # líneas (Ej. WC 25).
- WD # : Habilita una ventana de datos de # líneas (Ej. WD 15). Shift-F3 cambia el formato de los datos.
- WR : Habilita la ventana de registros.
- WW # : Habilita una ventana de vistas de # líneas (Ej. WW 5).
- ALTKEY KeyCombo : Cambia el hotkey de Soft-Ice (Ej. "ALTKEY ALT Z").
- ^ : El acento circunflejo ("^") hace que un comando sea ejecutado sin mostrarse en la línea de comandos.
- ; : El punto y coma simula que el usuario presiona la tecla "ENTER".

Aquí hay algunas líneas "init" de ejemplo de WINICE.DAT:

- INIT="lines 60;code on;wd 13;wc 25;wr;wl;dex 1 ss:esp;faults off;color 80 8F 70 74 8F;X;"
- INIT="ww 5;watch es:di;watch eax;watch *es:di;watch *eax;color 17 1F 90 98 1A;X;" • INIT="set font 2;set origin 150 25;lines 50;wr;wl;wd 10;wc 20;color 07 0B 71 04 02;DEX 0 SS:ESP;X;"

Universal Video Driver: Soft-Ice 3.2 viene con un controlador de video universal que puede ser utilizado virtualmente con cualquier tarjeta de video. En lugar de cambiar entre modos gráficos de texto, Soft-Ice aparece como una ventana en el escritorio. Se le puede cambiar el tamaño a la ventana usando el comando "lines 25|43|50|60", puede ser movida por la pantalla usando las combinaciones "CTRL-ALT-Flechas de dirección" y puede ser mas legible usando "set font 0|1|2|3" para incrementar o decrementar el tamaño de la fuente (ten en cuenta que puede hacer falta cambiar el tamaño de la ventana usando el comando "lines" después de cambiar el tamaño de la fuente). Ten en cuenta que la localización de la ventana puede ser ajustada manualmente usando el comando "set origin x y" (x 150 y 25 = centrada en una pantalla de 800x600).

Internal Variables: Soft-Ice usa un número de variables internas para retener la información de los ajustes. Estas variables pueden ser cambiadas en forma dinámica usando el comando "set *variable parámetro*", e incluir lo siguiente:

- ALTSCR (on|off, mueve la pantalla a un segundo monitor, monocromo)
- CASESENSITIVE (on|off, hace que los símbolos locales o globales tengan sensibilidad a las Mayúsculas o Minúsculas)
- CODE (on|off, muestra los bytecodes hexadecimales)
- EXCLUDE (on|off)
- FAULTS (on|off, captura Fallas de Protección General y de Paginado)
- I1HERE (on|off, hace que cualquier INT1 embebida invoque a SoftIce)
- I3HERE (on|off, hace que cualquier INT3 embebida [Ej. de una llamada DebugBreak() de la API Win32] invoque a SoftIce)
- LOWERCASE (on|off, muestra el código en minúsculas)
- MOUSE (on|off 1|2|3, habilita el mouse, ajusta la velocidad del mouse de 1 (lento) a 3 (rápido))
- PAUSE (on|off, realiza una pausa después de cada vez que se llena la pantalla en la ventana de comandos)
- SYMBOLS (on|off , usa la información de los símbolos)
- TABS (on|off 1|2|3|4|5|6|7|8 , ajusta el TAB cada 1-8 caracteres)
- THREADP (on|off)
- VERBOSE (on|off, muestra mensajes de diagnóstico como "module load/unload xxx")

Los ajustes actuales pueden ser vistos escribiendo "set variable" sin parámetros.

Ajustes de Winice.dat Algunas variables pueden ser ajustadas en winice.dat sin usar la línea "init". Estas incluyen los siguientes comandos. (cortesía de THE OWL):

- NOPIC2=on|off
- NOLEDS=on|off (habilita/deshabilita la programación de capslock y numlock)
- NOPAGE=on|off (habilita/deshabilita el mapeo de páginas no presentes)
- NOCONTEXT=on|off
- MACROS=*number* (número máximo de macros)
- NMI on|off(captura Non-Maskable Interrupt-útil con hardware breakout switch)
- VERBOSE=on|off (habilita/deshabilita mensajes de diagnóstico)
- KBD=on|off (parchea/no parchea el controlador de teclado)
- LOWERCASE=on|off (habilita/deshabilita assembly en minúsculas)
- MOUSE=on|off (habilita/deshabilita el soporte de mouse)
- THREADP on|off (habilita/deshabilita pasos específicos del thread)
- EXCLUDE (parámetro de rango)
- SIWVIDRANGE on|off
- TRA=*tamaño* (tamaño del buffer de trace)
- HST=*tamaño* (tamaño del buffer de historia de comandos)
- LOAD=*archivo* (carga símbolos)
- LOAD%= *archivo*
- LOADX= *archivo*
- LOADX%= *archivo*
- LOAD32= *archivo*
- LOAD32%= *archivo*
- PHYSMB=*tamaño* (cantidad de RAM del sistema)
- PHONE=*número* (número telefónico para depuración serial)
- COM1=on|off (habilita el COM1 para depuración serial; funciona para COM2 y COM3
- DINIT=*cadena* (DISCA la cadena de inicialización para depuración serial)
- ANSWER=*cadena* (CONTESTA la cadena de inicialización para depuración serial) • EXP=*archivo* (carga exports)
- VDD
- DDRAW
- DRAWSIZE=*tamaño* (tamaño de la memoria de video en K)
- MONITOR
- WDMEXPORTS
- DISPLAY=*tipo* (*tipo* es VIPE, S3, MACH32,MACH86, 0, o VD)
- FAULTS=on|off (captura FPGs - Fallas de Protección General)
- SYM=*tamaño* (tamaño de memoria reservado para símbolos)
- LBRECORD=on|off (habilita/deshabilita la colección de mensajes de la última rama --last-branch message collection)

Los macros pueden ser definidos en winice.dat ingresando la línea 'MACRO *nombre*=''*comandos*'' (por ejemplo, MACRO Qexp="addr explorer; Query %1"). Finalmente, los comandos de teclas de función pueden ser ajustados usando la sintaxis *key_combo*="^*comando*;", como por ejemplo SF1 = "^watch eax;" (ajusta Shift-F1 a "watch eax");. En esta sintaxis, A significa ALT, C significa CTRL y S significa SHIFT

Traceando a través del código

• BPRW *modulo/selector* [T|TW] [*condiciones...*] [vii] La idea de un depurador es que se puede ir paso a paso a través del código del programa y ver los cambios que se realizan a los registros, memoria, variables y banderas (flags) que el programa utiliza. En general, querrás un "Program Step" o paso de programa usando la tecla F10, salteando llamadas irrelevantes y rutinas de servicio del Sistema Operativo (SO). Cuando encuentres una sección de interés, y cuyo código quieres desensamblar a conciencia (la mayoría de las veces una parte del listado de ASM del desensamblador que necesita clarificarse[iv]), vas a dar "Single Step" o pasos simples a través del código usando la tecla F8 para ver exactamente que está haciendo el programa. Es importante tener los export symbols cargados para las .DLL de Windows[v], ya que a menos que estés depurando el kernel, no quieres perder la pista de una llamada API estándar que pasa por varias funciones y subfunciones de USER.EXE. También puedes quedar atrapado en una llamada de la que quieres salir desesperadamente, para lo que puedes usar la tecla F12 para saltar hasta la próxima instrucción RET.

• P : Ejecuta un paso de programa, ejecutando las llamadas como una instrucción simple. Atajo: la tecla F10.

• P RET : Ejecuta hasta que se encuentre una instrucción return. Atajo: la tecla F12.

• T : Ejecuta una instrucción. Atajo: La tecla F8.

Modo Back Trace

El modo Back Trace de Soft-Ice es malentendido a menudo, usualmente a causa de su nombre. El modo Back Trace no tracea hacia atrás a través del código, en realidad mantiene un buffer de instrucciones que son almacenadas cuando se dispara un breakpoint (con el calificador T o TW). El usuario entonces puede ver o tracear a través de los comandos en el buffer[vi] (Ej: los registros que no cambiaron, las direcciones fuera del buffer a las que no se puede saltar o llamar). El modo Back Trace es controlado por cuatro instrucciones principales:

• BPR *inicio fin* [T|TW] [*condiciones...*] : El comando BPR puede ser usado con T o TW para ajustar un breakpoint que disparará una escritura al buffer de trace cuando una dirección de memoria es ejecutada o escrita. Ingresando BPR con las direcciones de inicio y final del rango de memoria, seguido de T (para Back Trace en ejecución) o TW (para Back Trace en escritura de memoria) y encapsulado por cualquier condición para el Breakpoint, se instalará un Breakpoint de rango de memoria que, en lugar de detener la ejecución, escribirá código al buffer de trace. Este buffer puede ser visto con el comando SHOW, o traceado usando el comando TRACE.

• BPRW *modulo/selector* [T|TW] [*condiciones...*] [vii]: El comando BPRW puede ser usado con T o TW para ajustar un breakpoint que disparará una escritura al buffer de trace cuando un módulo de Windows o selector es ejecutado o escrito. Ingresando BPRW seguido del

nombre del módulo o del selector a ser capturado, seguido por T (para Back Trace en ejecución) o TW (para Back Trace en escritura) y encapsulado por cualquier condición para el Breakpoint, se instalará un Breakpoint de rango de memoria que escribirá código al buffer de trace *en lugar* de detener la ejecución (parecerá que el programa no se detiene, sin embargo esto es una ilusión). Este buffer puede ser visto con el comando SHOW, o traceado con el comando TRACE. Ten en cuenta que usar la función BPRW hará mas lento tu sistema en forma considerable (por ejemplo, inténtalo con el módulo "kernel").

- SHOW *inicio* [l *longitud*] : El comando SHOW muestra los contenidos del buffer de trace a partir del número de índice especificado en *inicio* para el número de instrucciones dado en *longitud*. Ten en cuenta que index 1 muestra la instrucción mas reciente en el buffer. Las instrucciones son mostradas en la ventana de comandos, y pueden desplazarse usando las flechas hacia arriba y abajo. ESC sale del modo SHOW.
- TRACE off | *index* : El comando TRACE es usado para tracear pasos a través del buffer de Back Trace comenzando en la instrucción especificada en *index*. Ten en cuenta que un valor de index de 1 indica la instrucción más reciente en el buffer, mientras que un valor de index de B (no 0X0B) indica el valor mas viejo en el buffer de trace. El buffer puede ser traceado usando XT, XP y XG en lugar de los comandos T (F8), P (F10) y G (Nota: Si usas Back Trace frecuentemente, sería bueno que configuraras las siguientes teclas: AF8="^XT;", AF10="^XP;", and AF12="^XP ret;"). El comando TRACE OFF es usado para salir del modo Back Trace.

Ver y Editar la Memoria

A medida que traceas por el código, a menudo vas a necesitar ver los contenidos de posiciones de memoria para mantener el control de variables, punteros e interrupciones. Además, puedes necesitar buscar un dato específico en el rango de memoria direccionable completo (por ejemplo, digamos, un nombre de usuario) para ajustar un Breakpoint en acceso a ese dato. Los contenidos de cualquiera de las ventanas en la pantalla de Soft-Ice pueden ser editados accediendo a la ventana con una combinación de teclas ALT (ALT-R para registros, ALT-D para datos, ALT-C para código, etc) y modificando directamente los datos mostrados.

- D : Muestra los contenidos de memoria. Sintaxis: **D** *tamaño dirección* **L** *longitud*, donde longitud puede ser B (byte), W (word - palabra), D (double word - palabra doble), S (short real - entero corto), L (long real - entero largo) o T (entero de 10 bytes), y dirección es la dirección de memoria a ser mostrada (una dirección apuntada por los contenidos de un registro -como EAX- o un desplazamiento u offset - como [EPB-04] puede ser mostrada escribiendo el nombre del registro y

el desplazamiento en lugar de la dirección). La l muestra los datos en la ventana de comandos (el área de muestra por defecto es la ventana de datos) y longitud determina cuantas filas (a cuatro bytes cada una, como en la ventana de datos) de memoria mostrar. Ten en cuenta que hay cuatro ventanas de datos, que pueden ser alternadas escribiendo DATA en la línea de comandos.

- DEX : Asigna una expresión a una ventana de datos. Sintaxis: **DEX** *número-de-ventana* [*expresión*], con número-de-ventana siendo un número de 0 a 4 y expresión siendo una expresión que resulte en una dirección de memoria. Ejemplo: **DEX 1 ss:esp** ajusta DEX0 para mostrar la parte superior de la pila, **DEX 1 @nombre-de-variable** ajusta DEX1 para mostrar las direcciones de memoria apuntadas por nombre-de-variable (tales como [ebp+10]), **DEX 1 @eax** ajusta DEX1 para mostrar el rango de memoria apuntado por eax. Una ventana de datos puede ser seleccionada con el comando DATA *número-de-ventana* (Ej: DATA 0, DATA 1, DATA 2 y DATA 3) o se puede alternar entre las ventanas usando el comando DATA.
- S : Busca en la RAM una cadena. Sintaxis: **s** *dirección* **L** *longitud lista-de-datos*, siendo dirección la dirección de memoria donde comenzar a buscar, longitud la longitud en bytes del área de búsqueda (FFFFFFFF es el máximo) y data-list el dato que estas buscando (las cadenas deben estar entre comillas simples o dobles. Los bytes son simplemente los bytecodes hexadecimales). Modificadores: -c hace que la búsqueda ignore si es mayúsculas o minúsculas (case-insensitive), -u busca una cadena Unicode. **S 30:0 L ffffffff 'username'** busca la cadena "username" en los 4 GB completos de espacio de direcciones. **S es:di 10 L ecx 'Hello',12,34** busca la cadena "Hello" seguida de 12h y 34h en el espacio de direcciones que comienza en es:di, por el número de bytes especificado en bytes (en hexa) en el registro ECX.[viii]

Vistas

Las vistas te permiten mantener la pista de una variable mientras estas depurando un programa. No es necesario decir que es una función importante en el crackeo y la ingeniería inversa. Para abrir una ventana de vistas, escribe ww en la línea de comandos de Soft-Ice. Si escribes **watch** seguido de un nombre de variable (Ej: **watch user_id**) agregas esa variable (y su valor) a la ventana de vistas. Los registros y desplazamientos de pila (para no mencionar sus valores de memoria) pueden ser vistos usándolos en lugar del nombre de variable, por ejemplo **watch es:di** y **watch [ebp 18]**. Para agregar, dado que muchos registros y desplazamientos de pila simplemente apuntan a una dirección donde se almacenan las variables reales, puedes ver los valores referenciados por ellos escribiendo un asterisco antes del nombre del registro o desplazamiento (Ej: **watch *es:di**). Unas buenas variables para ver son es:di, *es:di, eax y cualquier [esp ?] o [ebp ?] que referencia entrada por un usuario.

Usando Breakpoints

Es imposible depurar en forma efectiva sin el conocimiento del trabajo de breakpoints (o puntos de quiebre). Un breakpoint es simplemente una instrucción a la CPU de detener la ejecución en el momento de acceder a cierta área de memoria o en un evento del programa o del sistema (tales como mensajes de Windows o instrucciones del programa) y luego entregarle la ejecución al depurador. Cuando ajustas un breakpoint en Soft-Ice y corres el programa en cuestión, Soft-Ice aparecerá cuando se encuentre la condición para el breakpoint, con su pantalla usual de código, registro y datos, detenido en la instrucción que causó que se active el breakpoint.

- **Bc # : Clear Breakpoint--elimina un breakpoint ajustado. Debes conocer el número de breakpoint.**
- **Bd # : Disable Breakpoint--deshabilita pero no elimina un breakpoint ajustado en Soft-Ice. Debes conocer el número de breakpoint.**
- **Be # : Enable Breakpoint-Habilita un breakpoint que ha sido deshabilitado. Debes conocer el número de breakpoint.**
- **Bl : List Breakpoints-Muestra una lista de todos los breakpoints ajustados actualmente en Soft-Ice, su estado y su número de breakpoint.**
- **Bmsg : Quiebre en mensaje de Windows.**
  Sintaxis: BMSG *window handle* L *begin-message end-message*
- **Bpint : Quiebre en Interrupción. Solo funciona con interrupciones manejadas por el IDT (95/NT).**
  Sintaxis: BPINT *numero-int*
- **Bpio: Quiebre en lectura y/o escritura de un puerto de Entrada/Salida.**
  Sintaxis: BPIO *puerto* [R|W|RW] [EQ|NE|GT|LT|M *valor*] [c=*cuenta*]
- **Bpm: Quiebre en lectura, escritura o ejecución de memoria.**
  Sintaxis: BPM[B|W|DW] *dirección* [R|W|RW|X]
- **Bpx : Quiebre en ejecución.**
  Sintaxis: BPX *dirección/símbolo* [c=*cuenta*] [EQ|NE|GT|LT|M *valor*] [c=*cuenta*]
- **Bpr : Quiebre en Rango de Memoria.**
  Sintaxis: BPR *dirección-inicio dirección- fin* [R|W|RW|T|TW] [*expresión*]
- **Bprw : Quiebre en segmento de programa/código.**
  Sintaxis: BPRW *nombre- módulo|selector* [R|W|RW|T|TW] [*expresión*]
- **CSIP : Calificar breakpoints. Sintaxis: CSIP *[off |[not] dirección-inicio dirección-fin | nombre-módulo]* Usar *not* hace que el breakpoint ocurra solo si el CSIP no es igual a las direcciones de inicio o módulos de windows listados. De otra forma, la ejecución se detiene solo si los breakpoints especificados por el usuario se encuentran dentro de la dirección o módulos especificados. "Off" deshabilita el control de CSIP.**

Usando Símbolos y NMSYM

***De un archivo .EXE*: Crea un archivo .MAP con IDA, o un archivo .SYM con BUILDSYM.EXE (BUILDSYM *nombre del archivo exe*)**

***De un archivo .DBG*: Lo convierte a archivo .MAP o .SYM usando DBG2MAP.EXE (DBG2MAP /m *archivo debg*)**

***De un archivo .MAP*: Convierte el archivo .MAP a archivo .SYM con MAPSYM.EXE (MAPSYM -mn *archivo map*), MSYM.EXE (MSYM *archivo map*) o TMAPSYM.EXE (TMAPSYM *archivo map*)**

***De un archivo .SYM*: Convierte el archivo .SYM a archivo .NMS con NMSYM.EXE (NMSYM /TRANS:SYMBOLS *archivo sym* /OUT: *archivo.nms*) NMSYM.EXE tiene las siguientes opciones de líneas de comando:**

- **/TRANSLATE o /TRANS (Traduce Símbolos [PUBLIC, TYPEINFO, SYMBOLS, SOURCE], ej. "/TRANS:SYMBOLS target.exe")**
- **/LOAD (Carga/ejecuta un modulo, ej. "/LOAD:NOBREAK target.exe", "/LOAD:Execute target.exe")**
- **/SOURCE (Nombre del archivo de origen, ej. "/TRANS:PACKAGE /SOURCE:target.exe")**
- **/ARGS (Especifica los argumentos del módulo cargado, ej. "/LOAD:Execute /ARGS:test.txt notepad.exe")**
- **/OUTPUT o /OUT (Nombre del archivo de salida, ej. "/TRANS:PUBLIC /OUT:target.nms")**
- **/PROMPT (Solicita los archivos fuente faltantes)**
- **/SYMLOAD o /SYM (Carga Símbolos, ej. "/SYM:kernel32.nms;user32.nms;target.exe")**
- **/EXPORTS o /EXP (Carga exports, ej. "/EXP:user.exe;kernel32.dll;msvcrt42.dll")**
- **/UNLOAD (descarga símbolos o exports, ej. "/UNLOAD:user.exe;kernel32.dll;msvcrt42.dll")**
- **/LOGFILE o /LOG (Graba el historial de comandos, ej. "/LOG:sice.txt", "/LOG:sice.txt,APPEND")**
- **/VERSION o /VER (Versión del Producto)**
- **/HELP o /H (Ayuda)**

La Utilidad Map2Map: Gij escribió una utilidad para convertir los archivos IDA .MAP a un formato adecuado para NMSYM.EXE; se llama Map2Map y es necesario Perl.

El Historial de Comandos del Soft-Ice: El historial de la ventana de comandos del Soft Ice puede ser guardada en cualquier momento usando el Symbol Loader o NMSYM.EXE. Los contenidos enteros del buffer del historial (de un tamaño determinado en WINICE.DAT) son guardados al archivo especificado. Ten en cuenta que sólo el texto que aparezca en la ventana de comandos será guardado. Los datos pueden ser volcados desde la ventana de comandos usando la orden "D" (Dump - Volcar) y especificando una longitud ("l") para volcar y el código puede ser volcado a la ventana de comandos usando la orden "U" (unassemble - desensamblar) y especificando, igualmente, su longitud ("L"). Ten en cuenta que si vas a grabar el historial de la ventana de comandos, sería aconsejable poner en WINICE.DAT lo siguiente: VERBOSE=OFF .

Opciones de la Línea de Comandos

Cortesía de THE OWL

- **/? (muestra la pantalla de ayuda)**
- **/tra *size* (Cantidad de historial de Back Trace en K decimales)** • **/x**
- **/m**
- **/nmi on|off (Capturar NMI - Non-Maskeable Interrupt)**
- **/nol on|off (Sin programación de Caps-Lock/Num-Lock)**
- **/vdd**
- **/kbd on|off (Parchar controlador del teclado [Caps/Num-Lock])**
- **/pen on|off (Habilitar soporte Pentium)**
- **/com[123] (Usar puerto COM 1-3 para depuración serial)**
- **/hst *size* (Cantidad de memoria adicional del historial de pantalla en K decimal)**
- **/exp *nombre de archivo* (Carga exports de nombre de archivo)**
- **/l *nombre de archivo* (Carga la información de símbolos de nombre de archivo)**
- **/l% *nombre de archivo* (Carga la información del símbolo de nombre de archivo)**
- **/load *nombre de archivo* (Carga la información de símbolos de nombre de archivo)**
- **/load% *nombre archivo* (Carga la información de símbolos de nombre archivo)**
- **/loadx *nombre de archivo* (Carga la información de símbolos de nombre de archivo)**
- **/loadx% *nombre archivo* (Carga la información de símbolos de nombre archivo)**
- **/load32 *nombre archivo* (Carga la información de símbolos de nombre archivo)**
- **/load32% *nombre archivo* (Carga la información del símbolo de nombre archivo)**
- **/sym *size* (Cantidad de memoria de tabla de símbolos en K decimal)**

Obteniendo Información de Sistema

- Addr : Muestra o Cambia a un Contexto de Dirección
- Class : Muestra información de Clases de Windows (Windows Classes).
- CPU : Muestra los registros de la CPU.
- Exp : Muestra o carga los exports de archivos DLL.
- GDT : Muestra la tabla de descriptores global.
- Heap : Muestra el acumulador global de Windows.
- Heap32 : Muestra/avanza el acumulador global de Windows.
- HWND : Muestra información sobre Windows Handles.
- IDT : Muestra la tabla de descriptor de interrupciones.
- LDT : Muestra la tabla de descripción local.
- LHeap : Muestra la Pila Local de Windows.
- Map32 : Muestra un mapa de memoria de todos los módulos de 32-bits cargados en memoria.
- MapV86 : Muestra el mapa de memoria de DOS de la Máquina Virtual actual.
- Mod : Muestra la lista de módulos de Windows.
- Page : Muestra información de la tabla de página.
- Proc : Muestra información sobre un proceso.
- Stack : Muestra una pila de llamada.
- Sym : Establece o Muestra un símbolo.
- Task : Muestra lista de tareas de Windows.
- Thread : Muestra información sobre un thread.
- TSS : Muestra el Segmento de Estado de Tarea y conexiones de los puertos de Entrada/Salida.
- VCall : Muestra los nombres y direcciones de memoria de las rutinas llamables en VxD
- VM : Muestra información de máquinas virtuales.
- VXD : Muestra mapa VxD de Windows.
- VMM : Llama al Menú de Servicios de Información del Depurador VMM.
- VPICD : Llama a Menú de Información del Depurador VPICD.
- VXDLDR : Muestra información de VxD.
- WMSG : Muestra los nombres y número de mensaje de los mensajes de Windows.

Capturando Fallas de Protección General (FPG)

Esta sección está todavía en fase de "experimentación". Soft-Ice captura todas las FGPs por defecto, pero es poco consuelo, ya que no hay mucho que podamos hacer al respecto. He advertido dos patrones de FGPs detectadas por Soft Ice: uno en el cual está atrapado en un loop Wait_Semaphore, y el otro en el cual todos los opcodes en la ventana de código son "FFFFFFFF INVALID". En el primer caso, puedes regresar con F12 hasta el código principal del Kernel y tratar de "sentir el código" (esto en "lenguaje cracker" significa ir hasta el final una y otra vez hasta que

veas algún patrón) y ver cual es el Jcc (salto condicional, por ejemplo JNZ) que causa el problema, entonces editar las banderas (R FL Z en Soft-Ice para cambiar de estado a ON o a OFF la bandera Zero) antes de ese JNZ particular (Esto me ha funcionado solo una vez, y aun así todo lo que hizo fue que mi computadora no se quedara colgada; la aplicación se colgó igual). En el otro caso, puedes (¡Rezar!, …es broma) poner la sentencia RET usando el comando **A** del Soft Ice (**A** *dirección,* donde *dirección* es la línea con el código FFFFFFFF INVALID) y esperar que se haya llegado a esta área de código mediante una llamada (CALL) y no un salto (esto no me ha funcionado de ninguna forma, pero la teoría es más o menos correcta;). A menos que hayas escrito y/o memorizado el código fuente del Kernel de Windows, vas a tener que vivir con las FGPs (pero debes caer luchando!).

Ejercicios

Soft-Ice es un programa sumamente poderoso con una interfaz complicada, en el que lo mejor es aprender por medio de la experiencia. Los siguientes ejercicios están hechos para familiarizarse con el Soft Ice.

Ejercicio 1: Depurando una aplicación existente.

A menudo nos frustraremos cuándo usemos un programa escrito por una tercera persona (tercera persona en este caso significando "no tu mismo", mas que "no Microsoft"); Soft-Ice es una herramienta excelente para resolver tales situaciones. Inso Corporation's Quick View Plus, por ejemplo, tiene un bug o error en la ventana "Register": cuando introducimos un número de serie y pulsamos "Register", el programa nos muestra un mensaje diciendo que el número que hemos introducido es inválido. Por supuesto, Inso Corp ha calificado este bug como una "medida de seguridad", pero nosotros estamos aquí para arreglarlo.
Comenzaremos instalando el Quick View Plus Trial edition; El Install Wizard nos guiará a través de sus deprimentes pantallas azules y sus ventanas de diálogo grises. Cuando termina, Quick View Plus se ejecutará y una pequeña ventana gris con tres botones ("Purchase", "Uninstall", "Continue") aparecerá; es aquí cuando el ejercicio comienza en serio.

1) Haz click en "Purchase", luego en "$49", después en "Accept", y finalmente en "Unlock by Phone".

2) Pulsa Ctrl-D para ir al Soft Ice. Vamos a intentar localizar los mensajes de la ventana de registro ("Register"), y para lograr esto necesitamos tantear un poco el terreno: BMSG requiere un parámetro hwnd, y la orden HWND requiere un nombre de proceso. Por tanto primero escribiremos TASK en la línea de comando del Soft Ice, el cual nos muestra lo siguiente:

```
Taskname       SS:SP       StackTop    StackBot   StackLow   TaskDB  hQueue  Events
Order32        0000:0000   005ED000    005F0000               11DE    2EEF    0000
...
```

Aquí está, arriba a nuestra izquierda: Order32. La sintaxis para HWND es HWND *tarea*, entonces ahora pondremos HWND ORDER32, que dará la siguiente salida:

```
Window-Handle   hQueue   SZ   QOwner    Class_Name          Window Procedure
07A8(1)         2EEF     32   ORDER32   #32770 (Dialog)     173F:00004757
07AC(2)         2EEF     32   ORDER32   Static              173F:000052FA
.......
 07DC           2EEF     32   ORDER32   Edit                173F:00000BF4
```

Nos lista demasiadas ventanas, siendo 07A8 la ventana de diálogo principal, pero la única que nos interesa está en el control de edición: 07DC.

3) Lo que vamos a hacer ahora es colocar un breakpoint en la ventana 07DC (el control de edición (Edit)) para el mensaje de Windows WM_GETTEXT; localizando solo este mensaje (el cual recibe el texto de un control de edición), evitamos tener que andar a través de las distintas rutinas de dibujo de pantalla y de manejo del mouse que Windows procesa cada millonésima parte de segundo. Ten en cuenta que en lugar de eso podríamos poner un breakpoint en GetDlgItemText, GedDlgItemTextA, GetWindowText, y GetWindowTextA para lograr el mismo efecto, pero es mucho mejor colocar un sólo breakpoint en lugar de 4 (además el programador podría haber escrito su propia ruina "Get Text", en cuyo caso estos breakpoints nos fallarían). El comando que escribiremos en el Soft Ice es BMSG 07DC WM_GETTEXT.

4) Ahora estamos listos para la acción. Introduce el Código de Desbloqueo que mas te guste (Mi preferido es 666111666) y pulsa "OK". ¡El Soft Ice saltará inmediatamente, en USER!BOZOSLIVEHERE -- esta es la idea que Microsoft tiene de los chistes, me imagino; pulsa F12 para salir (RET) de esa función y llegaremos a USER!BEAR498, F12 otra vez hasta USER!GLOBALGETATOMNAME, F12 otra vez hasta llegar al código PROT16, otro F12 nos lleva a SER!DIALOGBOXINDIRECTPARAM, y finalmente un F12 mas y llegaremos a KERNEL.Alloc. Esto es muy importante para tener en cuenta, en la mayoría de los casos donde el breakpoint nos lleva al código de Windows (y F11 te sacará solo de una sola capa o layer), nos encontraremos en KERNEL.Alloc justo antes de regresar al código de la aplicación. Una buena regla general es por tanto pulsar F12 como locos hasta llegar a KERNEL.Alloc, luego presionar F12 una vez más para volver a la aplicación.

5) Pulsamos F12 otra vez, y nos encontraremos en ORDER32!.text 39B9, con el siguiente código:

```
0137:004049B9 Call [User32!GetDlgItemTextA]
0137:004049BF LEA ECX,[EBP-68]
0137:004049C2 PUSH ECX
0137:004049C3 CALL 004047E2
0137:004049C8 ADD ESP,04
0137:004049CB TEST EAX,EAX
0137:004049CD JNZ 004047E2
```

004049BF LEA ECX,[EBP-68] es la línea de código en la que estamos; Call User32!GetDlgItemTxtA recién fue ejecutado. Mirando las líneas de código que hay después, te darás cuenta de que esto es todo lo que necesitamos: éste es un esquema *típico* TEST/JNZ. El Unlock Code ha sido guardado en EBP-68 por GetDlgItemTextA; es luego movido a ECX y empujado a la pila como único parámetro de la función en 004047E2. Cuando el programa regresa de la llamada, la pila es corregida (ADD ESP,04) y un valor booleano (1 o 0, correspondiendo a TRUE o FALSE) es almacenado en EAX por la función. El valor en EAX está siendo comprobado para averiguar si es verdadero o TRUE (1) y si es así, el programa sigue adelante hacia el código "unlock- code-correcto-ahora-registra-a-este-pobre". En código-pseudo-C, esto sería así:

```
CHAR UnlockCode;
BOOLEAN IsGood;
...
GetDlgItemText(hwnd 07DC, int UnlockCodeField, charbuf [EBP-68], int 8);
UnlockCode=[EBP-68]; IsGood=ValidateUnlockCode(UnlockCode); if (IsGood)
UnlockProgram();
else MessageBeep();
...
```

Ahora debemos asegurarnos que estamos en lo correcto acerca de este código. F8 para pasar la instrucción LEA, luego escribimos D ECX: En la línea de comandos del Soft Ice, en la ventana de datos veremos:

```
013F:005EF604 36 36 36 31 31 31 36 36-00 05 00 00 7F 16 85 63  66611166....^..c
```

OK, esto es que nuestro serial está a punto de ser manipulado. Miramos detenidamente, ¿qué vemos? Nuestro querido 666111666 (o sus primeros ocho dígitos, en realidad), siendo cargado en ECX como parámetro de 004047e2. ValidateUnLockCode(UnlockCode). F10 hasta llegar al JNZ, paramos ahí. Vemos lo que pone en JNZ 00404A29 (NO JUMP) Ahora cambiaremos el estado del flag Zero, para ponerlo en estado no-cero, esto lo haremos escribiendo R FL Z. Vemos que el código ha cambiado a lo siguiente: JNZ 00404A29 (JUMP). En este momento pulsaremos Ctrl-D para volver al programa y veremos si hemos acertado... aparece una ventana nueva: **Quick View PLus Unlocked** Thank you for purchasing, la-la-li-la-la.... Pulsa "OK", ¡esta lección ha terminado!, Para finalizar, presionamos Ctrl-D y escribimos BC * para quitar todos los breakpoints.

Ejercicio 2: Recuperando el Acceso Perdido.

Cuándo tienes el sistema configurado con seguridad máxima, es siempre un peligro olvidar la contraseña y por tanto perder el acceso a nuestros datos. La mayoría de las veces simplemente puedes realizar un ataque "de fuerza bruta" a los archivos encriptados (o a cualquier objetivo que te dé problemas), pero cuando no tocas la PC durante un rato y se activa el salvapantallas que olvidaste que habías instalado, protegido con una contraseña que hace mucho tiempo que no recuerdas, tendrás que reiniciar -y perder datos- para volver a tener acceso a tu máquina. Soft-Ice, sin embargo, es lo suficientemente poderoso para solventar estas situaciones difíciles.

Para comenzar, instala un salvapantallas con protección de contraseña (si no lo tienes ya) a través del Panel de Control en Windows 95. Espera a que se active, luego mueve el mouse para activar la ventana de diálogo donde pide la contraseña. Vamos a ver:

1) Introduce una contraseña cualquiera, luego **Ctrl-D** y pon un breakpoint en hmemcpy (**bpx hmemcpy**) -hmemcpy es la función del kernel usada por Windows para manejar cadenas en la memoria, por ejemplo cuando son metidas en una ventana de diálogo siendo comprobadas para verificarlas. Presiona **Ctrl-D** otra vez para regresar al protector de pantalla, pulsa sobre el botón OK. Soft-Ice saltará y aparecemos en KERNEL!LOGERROR 0123

2) **F12** hasta que lleguemos a Kernel.Alloc (10 veces), luego otra vez para entrar en el código de nuestra víctima, en este caso llamado (asombrosamente) PASSWORD!.text

Ten en cuenta que apunta inmediatamente a nuestro ejecutable víctima, password.cpl en el directorio windows\system. Password.cpl es una extensión de 37,376 bytes del Panel de Control - inmediatamente sabemos que los archivos .scr usan el applet (o el subapplet) de contraseñas del Panel de Control para proteger tu máquina. Password.cpl exporta las siguientes funciones:

```
0000   00001151 CPlApplet
0001   00003f3b  PPChangePassword
0002   00003eb9 PPGetPasswordStatus
0003   00004006 VerifyScreenSavePwd
```

Y (como en cualquier programa de Windows) importa unas cuantas adicionales, incluyendo estas:

```
MPR.dll
004e   PwdSetPasswordStatusA
0011   PwdChangePasswordA
0013   PwdGetPasswordStatusA
003f    WNetGetUserA
```

Esta víctima esta poniéndose cada vez más interesante… puede ser necesario catalogarlo con dificultad "Difícil"…

3) Mira detenidamente el código:

```
0137:7C45428F   CALL [7C4582BC]
0137:7C454295   TEST EDI, EDI
0137:7C454297   JNZ 7C4542B1
0137:7C454299   LEA EAX, [EBP-04]
0137:7C45429C   LEA ECX, [EBP-14]
0137:7C45429F   PUSH EAX
0137:7C4542A0   PUSH ECX
0137:7C4542A1   CALL 7C454536
0137:7C4542A6   TEST EAX, EAX
0137:7C4542A8   JZ 7C4542DE
0137:7C4542AA   MOV EAX, 00000001
0137:7C4542AF   JMP 7C454322
```

Los esquemas de protección de contraseña normalmente no guardan una copia desencriptada de la contraseña en memoria. En general, la entrada de datos del usuario se encripta usando el mismo algoritmo que el encriptador de la contraseña, luego las dos cadenas encriptadas son comparadas (sí, puede hacerse mucho mas complicado que esto, pero esa es la idea básica). Lo que esto significa es que si vas en busca de la *contraseña,* te esperan un montón de cálculos matemáticos. Si tan solo quieres obtener *acceso,* entonces es algo tan simple como una declaración CMP -- el acercamiento a "los cambios de estado de los flags ".

Si investigas un poco en el código, veras que hay dos clases de saltos: uno que te lleva al rango 7C4542xx (el JNZ en 7C454295 y el JZ en 7C4542A8), y el que te lleva al rango 7C4543xx (el JMP en 7C454322). Lo que parece -y ésta es una intuición que aparece después de haber visto toneladas de código parecido a este- es que tienes dos comprobaciones (tal vez una para el tamaño de la contraseña y otra para el contenido de la contraseña, sin embargo en este ejercicio el propósito de las comprobaciones es algo sin importancia), luego un EAX = 00000001 (en booleano, "VERDADERO") seguido por un salto a una posición *después de* los "resultados" de las comprobaciones.

Te encuentras, por lo tanto, que quieres fallar en las dos comprobaciones (P.ej.: no saltar) y al final que vaya a 7C454322. ¿Cómo sabes que hay que hacer? Por ahora, eso es una incógnita. Pero si sigues traceando, te encontrarás con que con la contraseña errónea tomarás esos dos saltos (P.ej.: Si pasas por el primero, el segundo te atrapará) y al final te llevará al mensaje de "contraseña inválida".

(4) Le das a **F10** hasta llegar a JNZ 7C4542B1. Aquí cambias el flag Zero (**r fl z**), luego **F10** hasta: JZ 7C4542DE. Otra vez, cambias el flag Zero, y pulsa **CTRL-D** para regresar al salvapantallas... el cual inmediatamente desaparecerá y tendrás acceso al escritorio. Asegúrate

de quitar el breakpoint hmemcpy (BC *) antes de salir...

¿La Parte Buena? Ahora sabes como romper un insignificante esquema de protección de contraseña del salvapantallas usando Soft Ice (y muchos mas esquemas similares usando tu imaginación). ¿La Parte Mala? Nadie en cuya máquina quieras entrar va a tener instalado el Soft Ice. Por supuesto, después de una ingeniería inversa completa, puedes escribir una utilidad para acabar con la protección del .scr o engañar al archivo .cpl, luego insértalo en el autorun.inf de un CD-ROM y espero que la computadora "víctima" tenga habilitado el "Autoplay"....

[i] Durante todo este tutorial se hará referencia a la versión 4.05 de Numega SoftIce para DOS/Win95/Win98.

[ii] Desde luego es una recomendación que suscribimos. Tener una copia de estos manuales, en especial del "Command Reference", nos permitirá conocer que hace exactamente una orden que no empleamos a menudo o como es su sintaxis.

[iii] Hasta la fecha los autores solo han encontrado dicha información en un programa.

[iv] Hace referencia al crackeo de un programa mediante la localización de una cadena conocida (ver pag 10).

[v] Como ya se ha mencionado mas arriba en este tutorial, el winice.dat que acompaña el paquete de Sice ya incluye todas esta líneas EXP precedidas de un ; basta quitar el ; y listo.

[vi] Se trata de una de las funciones avanzadas del SoftIce. Cuando identificamos en Wdasm una parte del código que nos interesa tracear hacia atrás, pero no sabemos de donde viene (no hay cartel Referenced by ...), ponemos un BPR ***rango*** (zona de la que pensamos puede provenir o todo el programa si no tenemos ni idea) y un bpx en la parte que hemos visto en el Wdasm. Al romper el Bpx podremos hacer un ***show*** identificando de donde parte la llamada (generalmente un jmp [eax+48] o cosa similar, ya que si no Wdasm daría la referencia)

[vii] Similar a anterior salvo que en vez de indicar un rango de direcciones de memoria el SoftIce dispone automáticamente el rango que ocupa el módulo o programa en ejecución que le indicamos.

[viii] Instrucción muy útil para el desempacado manual de ejecutable comprimidos, pero que cuelga el sistema con frecuencia.

Capítulo 6

Editores Hexadecimales: WinHex

por Imanol

WinHex fue el primer editor hexadecimal que utilicé, y es por ello por lo que suelo usarlo. No es tan potente como otros, pero su interfaz sencilla lo hacen idóneo para aprendices.

Instalación y descarga

El sitio oficial de WinHex se encuentra en http://www.winhex.com aunque también puede ser descargado desde otras páginas como tucows, softonic, cnet. En esa misma página se encuentra otro de los productos hermanos de winhex, el RAMCheat, supongo que ahora entenderéis por qué empecé utilizando el WinHex. Este editor es Shareware, y su única limitación es no poder guardar archivos mayores de 200kb, por ello es aconsejable o bien comprarlo, o bien buscar por la red alguna versión sin dicha limitación.

Una vez descargado tendremos un zip que al descomprimirlo nos mostrará una serie de archivos, entre ellos uno llamado winhex.exe. Este es el archivo que tenemos que ejecutar. También hay un archivo de instalación llamado setup.exe, pero su ejecución no añade ninguna función que nos interese, ya que nosotros sólo lo utilizaremos como editor hexadecimal, y no
para otras utilidades que se anuncian en la página web, como clonado de datos o borrado irreversible de datos.. Entonces hagamos doble clic sobre el archivo WinHex.exe y veamos que
es capaz de hacer.

Primeros pasos

Una vez arrancado el programa aparece una ventana en la que se puede elegir entre acceder a uno de los proyectos recientes, continuar con la ultima sesión, abrir un script personalizado o preinstalado o bien elegir una de las cuatro

opciones de arriba que consisten en abrir un archivo, abrir una unidad de disco, abrir Ram y abrir el contenido de una carpeta. Lo normal es que queramos abrir un archivo, para ver el contenido de éste y modificar lo que hayamos hecho con el debugger. También puede ser interesante abrir la ram para modificar parámetros en aplicaciones como juegos, etc. Si pulsamos sobre esta opción sale una lista con las tareas que estamos realizando y pulsando sobre la que queramos se puede analizar su contenido.

Checksums

Un checksum es un número característico obtenido para verificar la autenticidad de los datos. Al realizar cambios sobre los archivos, es posible que si los cambios no se contrarrestan entre si, el checksum cambie, produciendo un error en la ejecución del archivo. Para evitar esto el checksum se puede recalcular pulsando ALT+F2.

Explicación de las opciones

Modos de edición

Existen tres modos fundamentales de abrir un archivo en winhex: Por defecto, sólo lectura y edición directa. En el primer modo se crea un archivo virtual cuando abrimos el archivo y al guardar los cambios se guardan permanentemente sobre el original. En el segundo modo no se realizan cambios sobre los archivos, permanecen protegidos, mientras que el modo de edición directa es ideal para archivos de gran tamaño, pero hay que tener en cuenta que los cambios se van realizando directamente sobre el original, aunque no pulsemos el botón de guardar.

Barra de estado

Una vez abierto un archivo, a lo largo de toda la parte inferior de la ventana se extiende la barra de estado que muestra el número de página actual (pulsando sobre él se puede avanzar a las siguientes), la posición actual (offset), la conversión decimal de los valores actuales en hexadecimal, el principio y final del bloque actual y el tamaño actual del bloque en bytes. Como ya se ha indicado, pulsando sobre ellos se puede actuar sobre dichas opciones. Además pulsando con el botón derecho del ratón sobre la barra de estado permite copiar la información desde ahí al portapapeles, si pulsamos con el botón derecho sobre el offset se puede cambiar entre representación absoluta o relativa, ideal para actuar sobre registros de longitud prefijada. Si pulsamos en la tercera opción de esta barra con el botón derecho podemos copiar al portapapeles los cuatro valores hexadecimales en orden inverso cosa que puede ser útil para examinar punteros.

Editores varios

Winhex es capaz también de editar discos y ram. La función de editar discos tiene ciertas aplicaciones como la de editar el espacio en blanco de un disco para poderlo copiar y abrirlo para buscar datos perdidos que no hayan sido machacados y también se pueden copiar sectores. Por su parte el editor de ram permite ver el contenido en ram de diferentes aplicaciones y modificarlo, con el inconveniente de que los modulos de 16 bits sólo pueden ser editados desde máquinas con win95/98 y que los módulos de sistema de windows no pueden ser alterados.

Algunas teclas rápidas

Hay ciertas teclas o combinaciones de teclas que pueden resultar de interés. La selección se realiza con el ratón, con el botón izquierdo, sobre una selección de ratón se puede deshacer pulsando dos veces con el botón derecho.
La tecla TAB sirve para cambiar de modo hexadecimal a Texto y viceversa.
La tecla INS sirve para cambiar del modo de sobreescritura al de insertar.

El menú

El menú principal contiene muchas opciones que son evidentes. Explicaré las más interesantes

Menú búsqueda
Esta es una de las funciones que más utilizaremos, que será buscar cadenas hexadecimales o de texto. El menú búsqueda presenta las siguientes opciones:
Encontrar texto: Busca cadenas de hasta 50 caracteres.
Encontrar valores Hexadecimales: Busca una cadena de hasta 50 valores hexadecimales de dos caracteres.
Reemplazar texto: Reemplaza un texto dado por otro seleccionado.
Reemplazar valores Hexadecimales: Funciona igual que el comando anterior, pero con valores hexadecimales en vez de texto.
Búsqueda combinada: Es una búsqueda en dos archivos diferentes de dos valores pero en
un mismo offset. Como ejemplo esto se puede utilizar para examinar partidas guardadas. Por ejemplo sabemos que tenemos 5 vidas en una partida guardada, guardamos otra con 1 vida y aquí realizamos la búsqueda y veremos cual es el valor que cambia, luego lo podemos modificar y poner 10 vidas o las que queramos.

Creo que con esto es suficiente para saber manejar el programa para nuestros intereses. Si hay dudas acerca de otras funciones conviene utilizar la ayuda del mismo programa.

Utilización del Ultra Edit

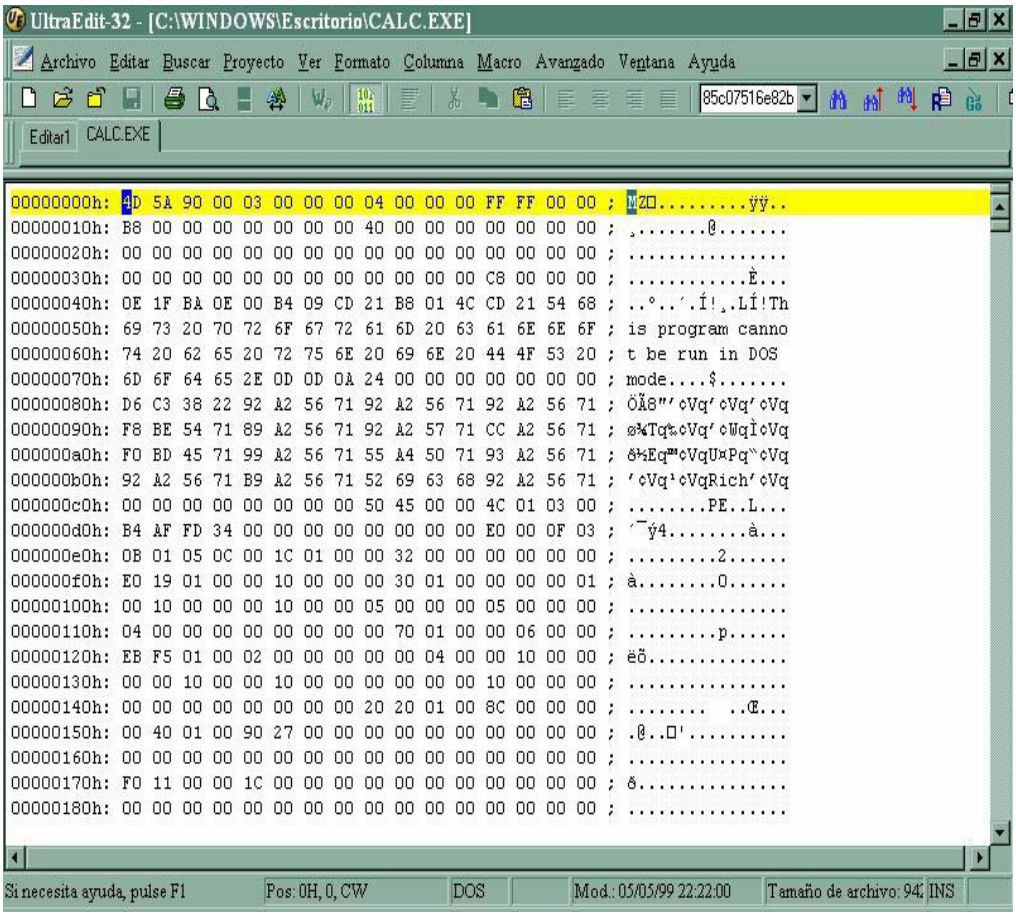
por Ricardo Narvaja

En esta lección trataremos de explicar el funcionamiento básico de ULTRA EDIT para ayudarnos en nuestra tarea de cracking ya que la cantidad de posibilidades que posee este programa son muchísimas y se puede consultar la ayuda que en las últimas versiones ya viene traducida al español, por si alguien quiere profundizar algún tema.

Primero que nada hagamos una copia del archivo que vamos a trabajar, esto es muy importante ya que una mala operación de este programa puede inutilizar un archivo por lo que SIEMPRE debemos trabajar sobre una copia y tener bien resguardado el original para poder restaurar.

Vamos a INICIO-BUSCAR y hacemos que busque calc.exe, una vez encontrado lo copiamos y pegamos en el escritorio para trabajar sobre esta copia dejando el original en su lugar sin tocar.

Abrimos el programa ULTRA EDIT (yo estoy trabajando con la versión 9 en español que hasta hoy es la ultima) y vamos a ARCHIVO-ABRIR y buscamos en el escritorio el calc.exe y se abre perfectamente,



Esta es una vista cuando ya abrimos el archivo, por supuesto como este programa es un editor hexadecimal veremos las posiciones de memoria en offset a la izquierda y su contenido en Hexadecimal en el centro, en la columna de la derecha vemos los caracteres ASCII correspondientes a cada posición de memoria si los hay, sino un puntito si no existe carácter para ese valor hexadecimal.

Lo primero que debemos saber es como ubicarnos en un valor offset para modificarlo.
Si nosotros utilizamos el Wdasm anteriormente y hallamos la posición de memoria que queremos modificar en la lección de Wdasm, se vio que en el borde inferior si estamos posicionados sobre la línea a modificar se puede leer el OFFSET de esa posición de memoria.

Pero que es exactamente el OFFSET que diferencia tiene con la posición de memoria?
En el desensamblado del Wdasm encontramos la respuesta allí en las primeras líneas podemos leer esto.

Number of Objects = 0003 (dec), Imagebase = 01000000h
Ese valor IMAGEBASE para que se den una idea es el comienzo desde donde se ubica el programa en la memoria, ojo no desde donde comienza a ejecutarse ya que puede comenzar por el medio del listado sino donde comienzan a encontrarse los primeros bytes en la memoria.

Fíjense en el Wdasm si bajan que las posiciones de memoria son todas superiores al valor de ImageBase.

Ahora si en el Wdasm estoy en esta posición 01001035

```
:01001035 76F7                  jbe 0100102E
:01001037 BFA86DF7BF                    mov edi, BFF76DA8
:0100103C E748                  out 48, ax
:0100103E F7BF0509FABF          idiv dword ptr [edi+BFFA0905]
```

y quisiera modificar estos bytes 76F7 en el Ultra EDIT como hago para encontrarlos allí?

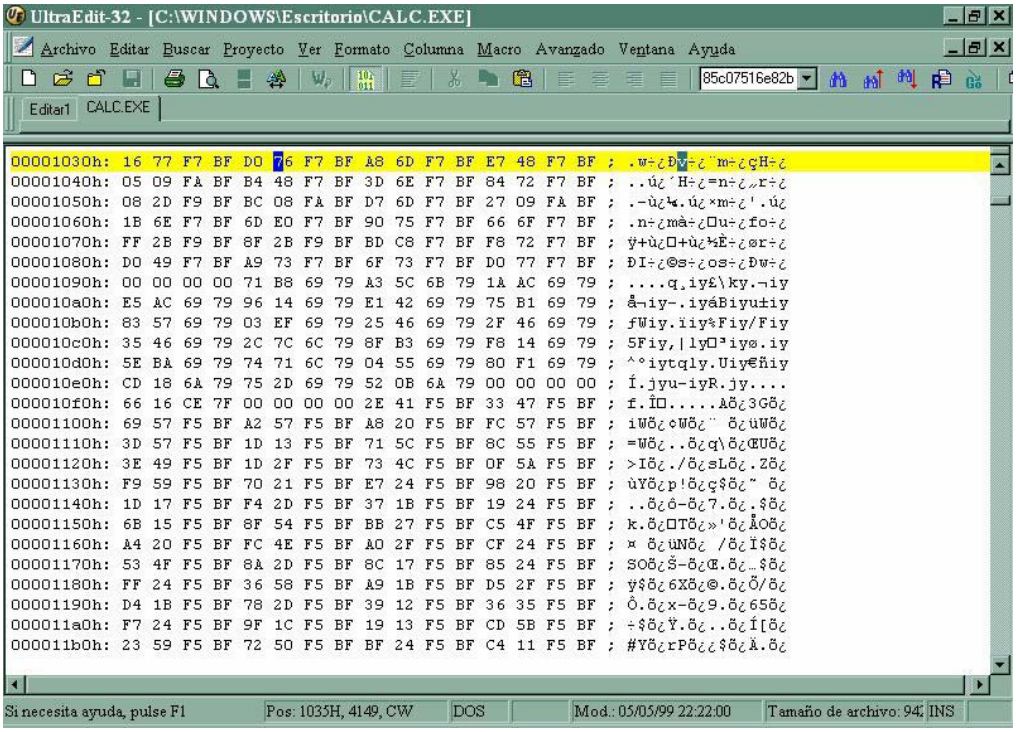
Si en el Wdasm me posiciono encima de esta sentencia leo en el borde inferior OFFSET que es 1035 por lo tanto nos damos cuenta en este caso fácilmente que el OFFSET es la posición de memoria menos la Image base o sea:

Posición de memoria	1001035
Image Base	1000000
resultado (offset)	1035

En este caso que la Image Base es 1000000 se calcula fácilmente pero en otros casos deberemos hacer la cuenta con la calculadora.

Bueno debemos encontrar los valores 76 f7 para modificarlos que se encuentran en el OFFSET 1035, vamos al ULTRA EDIT y allí vamos a

BUSCAR-IR A LINEA-PAGINA.
y en la ventanita escribimos 0x1035 ya que nos exige el 0x para demostrar que es un valor hexadecimal.



En la imagen podemos ver a la izquierda el valor 1030 del OFFSET y el valor 76 que buscábamos marcado en azul corresponde al offset 1035, los valores subsiguientes corresponden a los posteriores valores que encontrábamos en el WDASM, luego del 76 venia F7, y seguía en la sentencia de abajo en el Wdasm BF, A8 etcétera.

Por lo tanto hemos utilizado la forma de encontrar los valores del Wdasm en el Ultra EDIT por el método del OFFSET.

Otra forma aunque mas imprecisa y que puede llevar a error aunque a veces es mas rápida, es la de hallar la cadena, la menciono porque en muchos tutes se utiliza.

El método consiste en copiar del Wdasm a partir del valor 76 , una cadena de valores de 10 cifras por lo menos, en este caso seria

76 F7 BF A8 6D F7 BF E7

Vamos al Ultra Edit subimos la barra de desplazamiento hacia el principio del programa y allí marcamos la primera línea para buscar a partir de allí hacia abajo.

Vamos directamente a Buscar y allí ponemos la cadena en forma corrida y le damos a Buscar y allí la encuentra, siempre que usemos este método debemos

BUSCAR SIGUIENTE luego de hallar una coincidencia, para ver que no haya otra cadena igual mas adelante.

Ese suele ser el problema de este método que muchas veces se busca una cadena y se confía uno y modifica algo y resulta que había dos o tres cadenas iguales mas adelante que uno no verifico al no usar BUSCAR SIGUIENTE y modifica cualquier cosa y mal.

Siempre es aconsejable el método del OFFSET si se puede usar.
Bueno una vez que por cualquiera de los métodos encontramos los bytes a cambiar escribimos encima de los mismos los valores nuevos y vamos a ARCHIVOS-GUARDAR y listo ya estarán guardados los cambios y además el Ultra Edit generara un archivo de respaldo de extensión bak que en este caso se llamara calc.bak y que será original como era antes de realizar los cambios salvo con la extensión bak en vez de exe.

Este programa tiene muchísimas funciones y posibilidades mas que se pueden ver claramente en su manual, solo haremos mención a dos que nos pueden servir en el tema cracking.

La primera función nos da la posibilidad de elegir dos archivos para comparar y ver las diferencias, la misma se encuentra en el menú ARCHIVOS-COMPARAR ARCHIVOS, podemos poner la tilde en la opción SOLO MOSTRAR LINEAS QUE DIFIERAN y saldrán solo las líneas que hay diferentes entre los dos. Esto es útil para ver los cambios que uno realizo o cuando uno analiza un crack hecho por otra persona para ver que modifico el mismo al ejecutarlo (habiendo guardado una copia del original en otro lugar antes de aplicar el crack)

Otra función interesante es la de COPIAR Y AGREGAR que se encuentra en el menú EDITAR, y sirve para marcar una zona de valores hexadecimales, copiarla o cortarla y agregarla en otro lugar del mismo archivo a partir del lugar donde esta el cursor o también para agregarle a partir del cursor los valores hexadecimales copiados en otro archivo diferente

Con eso completamos esta visión básica del Ultra Edit en lo referido al tema cracking

TRABAJANDO CON HIEW

por Frantic

HIEW es un editor hexadecimal (hex editor) para DOS con el que se pueden abrir, observar y modificar archivos, generalmente, ejecutables (.exe) y librerías (.dll)

Los apuntes que siguen son una referencia de las funciones más útiles de este clásico del oficio, y están referidos a la versión 615.

Las ayudas que ofrece el propio programa (F1) se limitan a unos listados de teclas con escuetas definiciones.

Se recomienda seguir estos apuntes abriendo un archivo para poner en práctica lo que se va leyendo. Antes haz una copia del programa de prueba, por si resultara alterado por equivocación.

ABRIENDO UN PROGRAMA

Si se ejecuta HIEW desde su carpeta de Windows, se abre una ventana del DOS en la que aparece un cuadro de diálogo para localizar el archivo que se quiere abrir. También se puede iniciar HIEW de ese modo desde la línea de comandos, pero lo más práctico es enviar directamente a HIEW el archivo que se va a abrir. Para ello se puede colocar un acceso directo a HIEW en la carpeta SEND TO de WINDOWS, y enviar el archivo con la opción correspondiente del menú contextual (botón secundario del mouse). Para los más avezados en los intrígulis de Windows queda una opción parecida pero más rápida aún: incorporar directamente HIEW al menú contextual. Para ello hay que añadir la entrada correspondiente en el registro de windows.

(HKEY_LOCAL_MACHINE\SOFTWARE\Classes*\shell\Hiew)
(HKEY_LOCAL_MACHINE\SOFTWARE\Classes*\shell\Hiew\command)

Otra manera rápida de abrir un archivo con HIEW es arrastrarlo y soltarlo encima de un acceso directo a HIEW colocado en el escritorio.

SEARCH=BÚSQUEDA (F7)

Solo se pueden buscar cadenas de código hexadecimal. No busca, por tanto, su equivalente en ASCII. Por ejemplo, puede encontrar 85C0, pero no encuentra "test eax, eax", que es lo que aquello significa.

También se pueden buscar cadenas de texto. Esto último viene muy bien para localizar pantallas por su texto.

Para ir a la siguiente coincidencia de una búsqueda tenemos tres combinaciones: Ctrl+F7 , Ctrl+Enter o Mayúsc+F7.

GOTO = IR A (F5)

Teclea la dirección que te da el soft-ice pero con el punto delante (.0041192D). Así consigues ir a las direcciones locales (las que ves en Softice) en vez de las globales (las que ves en cualquier editor hex). Para bascular entre las variables globales-locales pulsar alt+F1

MODO (F4)

Ofrece tres modos. El más adecuado para analizar el código, como su nombre indica el es **Decode**. En ese modo se pueden realizar los cambios de las instrucciones (F3).

Para buscar cadenas de caracteres (texto) a simple vista (usando las teclas de desplazamiento: flecha arriba, abajo, pág arriba y abajo, control+inicio, control+fin), en cambio, es mejor pasar al modo Hex. Si, estando en modo Hex quieres realizar algún cambio (pulsando F3) se puede pasar de la parte central (códigos hexadecimales) a la parte derecha (caracteres ASCII) mediante la tecla de tabulación. Por cierto, para cambiar de modo sin pulsar F4, basta con pulsar Enter, y se van conmutando los tres modos de formar rotativa.

Hiew se puede inicializar directamente en modo "decode". Basta con abrir el archivo "hiew.ini" y cambiar el valor de 'StartMode' de 'Text' a 'Code'.

En los tres modos se puede guardar el contenido de la pantalla en un archivo, mediante la combinación Alt+P.

EDITAR (F3)

Es la tecla a usar para efectuar los cambios en un archivo. Antes debes situar el cursor en el primer carácter de la cadena a modificar. La posición del cursor se advierte porque está resaltada con otro color de fondo. Te puedes desplazar con las teclas habituales.
Cuando se pulsa F3 el cursor aparecerá en intermitencia. Puedes abandonar el modo edición pulsando la tecla Esc. Lo más habitual es editar en modo Decode, cuando se trata de efectuar cambios en los códigos hexadecimales. Cuando pulses F3, verás que cambia el sistema de direcciones; en lugar de tener en la columna de la izquierda las RVA (Direcciones Virtuales Relativas) que son las que nos aparecen en SoftIce y W32Dasm, se ven los desplazamientos... pero eso ya es otro tema. También te cambiarán los datos de la columna de la derecha por la misma razón. En suma, asegúrate bien de estar en la RVA que deseas antes de pasar al modo Edición para no confundirte con estos cambios. Ahora llegamos al punto en que HIEW tiene una de sus mejores bazas: puedes cambiar directamente el número hexadecimal o introducir la instrucción ASM que desees. Para esto último, una vez situado el cursor en el punto adecuado y tras pulsar F3, pulsas Enter (o F2) y te aparecerá una ventanita (Assembler) con la instrucción sobre la que vas a trabajar. Esto es muy cómodo porque permite realizar algunos cambios "a pelo" sin necesidad de saber qué números hex (opcodes) se requieren para ello. Por ejemplo, en la ventana Assembler puedes cambiar directamente una instrucción "push ebx" por "push eax" con solo escribirlo; luego HIEW se encargará de hacer aparecer los opcodes crrespondientes. Para editar las instrucciones ASM en esta ventana vienen muy bien las teclas "retroceso" "insertar" "supr" "inicio" "fin" y las flechas de dplazamiento. En cualquier momento puedes deshacer los cambios con la tecla Esc.

Cuando hayas terminado de hacer los cambios deseados, pulsas F9 para guardar los cambios, que ya no se podrán deshacer. Antes de guardar los cambios se pueden restablecer los caracteres originales situando el cursor en el lugar deseado y pulsando F3. Si ha salido algo mal, pulsa Esc para dejar el archivo como estaba.

El modo Hex también sirve para realizar cambios cuando, por ejemplo, quieres cambiar un texto que aparece en el programa, y has localizado el punto donde se encuentra. Caso típico suelen ser los epígrafes de las ventanas que rezan "Unregistered" o "Evaluation" que se quedan así aunque se hayan efectuado los cambios necesarios para evitar los trastornos que conllevan esas limitaciones. Bien, F5 para ir a la dirección deseada; F4 para pasar al modo Hex y F3 para editar. Ahora, pulsando la tecla de tabulación puedes pasar a la columna derecha (ascii) y realizar los cambios "a pelo" sobre los caracteres.

En este modo Hex hay unas teclas de edición de bytes (F4, F5, y F6) que despliegan el valor hexadecimal en formato byte, palabra y doble palabra respectivamente.

TERMINANDO

Con Esc se sale de HIEW. Si antes de salir se pulsa F10 se guardará la fecha actual como fecha de modificación del archivo (timestamp).

Capítulo 7

Nuestra primera víctima.

mR Gandalf

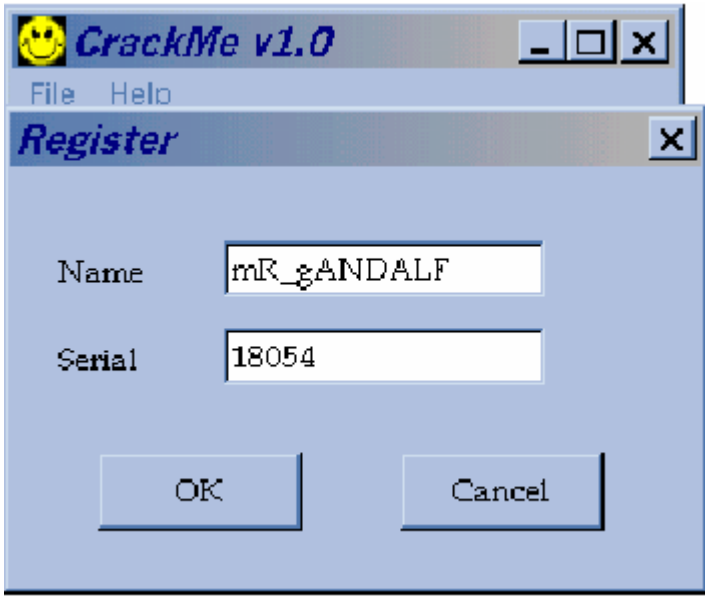
Una vez sentadas las bases teóricas ha llegado el momento de pasar a la acción. Todo lo necesario (y mucho más) para vencer la protección que utilizaremos como ejemplo ha sido extensamente expuesto en los capitulos anteriores. Nuesra misión no será solo la de cambiar el byte que invierte el comportamiento del programa dejándolo a nuestra merced. Iremos más lejos. Entenderemos que hace exactamente el programa y las distintas formas que tenemos de llegar a ese conocimiento. Estudiaremos su código y finalmente veremos como podemos hacer para que se comporte como queramos. La víctima elegida para nuestro primer crack es CrackMe v1.0 de Cruehead.

Un crackme es básicamente un programa hecho por crackers para crackers. Como su propio nombre indica, su única misión es la de ser craqueados y por lo tanto son una buena fuente material para aprender, si bien algunos son bastante complejos. El autor del ingenioso programita demuestra así su conocimiento de rutinas de protección y su habilidad como *coder* (todo buen cracker debe ser también buen coder) y el newbie aprende como evitar esa protección. Hay bastantes sitios de donde bajar crackmes. Basta teclear en Google "crackme" y nos saldrán un buen montón de páginas. Personalmente recomiendo los de la página de Karpoff. Ahí tenemos una amplia lista para los que se detalla su autor, algo sobre el tipo de protección que trata y si está resuelto o no.

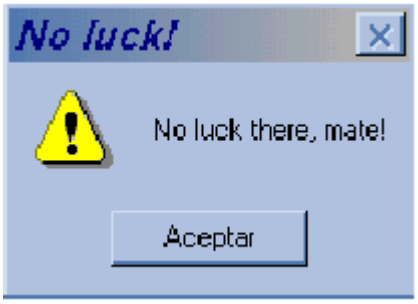
Entrando en materia

Con este programita Cruehead / MiB ilustran el más básico de los esquemas de protección. Fue posiblemente el primero de los que se usaron allá por los años 90, cuando aparecieron las dristibuciones shareware. Así pues, el programa recibe una cadena de caracteres que debe evaluar como cierta o falsa realizando después una determinada acción o consecuencia. Esquemáticamente el proceso consta de: Introducción del serial → encriptación → comparación → resultado. En cada uno de estos puntos podemos intervenir, siendo nuestro objetivo el conseguir que la aplicación funcione sin limitaciones. En su variante más primitiva la comparación se

realizaba sin ningún tipo de encriptación. El programa toma el serial que le introducimos, lo compara con el suyo, y si son iguales el usuario queda registrado y ya no tiene que volver a introducir nada. Sobre esta línea general pueden existir variaciones, por ejemplo, el programa guarda el serial que nosotros le damos y realiza la comparación al iniciar la próxima sesión, o varias veces durante la ejecución del mismo. A menudo lo que vemos al tracear es Call *rutina de comprobación* ; Test eax, eax (siendo aquí eax el *registro flag*) y JE *chico bueno* (en la terminología habitual, parte del programa que ejecuta sentencias que nos registran, en contrapartida a *chico malo*).

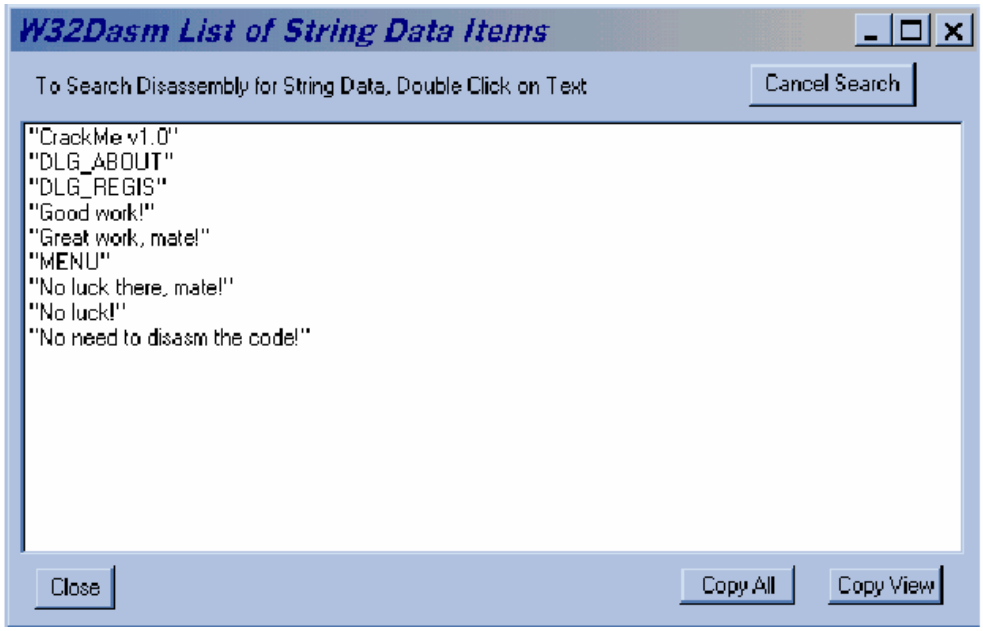


En el caso que nos ocupa, tenemos una caja de diálogo (DialogBox) que se abre clickeando sobre Help y luego sobre register. Nos pide un nombre y un serial. Si ponemos alguno al azar y pulsamos ok vemos el cartel de chico malo “No luck there, mate!”.



Búsqueda del núcleo de la protección por referencia a una cadena conocida

Aunque en próximos capítulo se explicarán otros métodos que se emplean aún antes de haber instalado y ejecutado por primera vez el programa, digamos que una de las primeras cosas que debemos intentar para vencer una protección es obtener su *listado muerto*, es decir, su código desensamblado. La mayoría de los programas ponen trabas a WDasm para que haga esto. También a IDA, a TurboDebbuger y a otros. Hay toda una serie de técnicas para superarlas pero, igual que en el caso anterior, son materia de futuros capítulos ;-). Por suerte esto es solo un crackme y sirve para aprender. Así que vamos a Wdasm, desensamblamos y echamos un vistazo en las String References, gesto que debe convertirse en habitual a partir de ahora. Aparece el siguiente cuadro de texto:



Aquí vemos algo que debe resultarnos familiar: “No luck there, mate!” Es la cadena de texto que aparece en la ventana cuando introducimos un serial al azar. Si clickeamos dos veces sobre esta línea Wdasm nos lleva hasta la zona del código donde es referenciada:

```
:00401361 C3                    ret

* Referenced by a CALL at Address:
|:00401245
|
:00401362 6A00                  push 00000000
```

```
* Reference To: USER32.MessageBeep, Ord:0000h
                  |
:00401364 E8AD000000          Call 00401416

:00401369 6A30                 push 00000030

* Possible StringData Ref from Data Obj ->"No luck!"
                  |
:0040136B 6860214000          push 00402160

* Possible StringData Ref from Data Obj ->"No luck there, mate!"
                  |
:00401370 6869214000          push 00402169
:00401375 FF7508              push [ebp+08]

* Reference To: USER32.MessageBoxA, Ord:0000h
                  |
:00401378 E8BD000000          Call 0040143A
:0040137D C3                  ret


* Referenced by a CALL at Address:
|:0040122D
|
:0040137E 8B742404            mov esi, dword ptr [esp+04]
:00401382 56                  push esi

* Referenced by a (U)nconditional or (C)onditional Jump at Addresses:
|:00401392(U), :0040139A(U)
```

Como podemos ver en este explícito trozo de código, el cartel que aparece cuando fallamos es un MessageBoxA. Si tomamos referencia de la API MessageBoxA (a mí me gusta tener a mano una copia de las Crackers Notes) comprenderemos exactamente que hace:

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

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

Returns

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

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

Si una caja de mensaje tiene un botón de Cancelación, la función devuelve el valor IDCANCEL[4] si la tecla ESC es apretada o si de botón de Cancelación es seleccionado. Si la caja de mensaje no tiene ningún botón de Cancelación, apretar ESC no tiene efecto.

Recordamos que en la convención stdcall que es la que maneja Win32 (salvo para svprintff()) los parámetros se pasan de derecha a izquierda, o sea:

```
:00401369 6A30                  push 00000030 <- determina el tipo de icono, el número
y el tipo de botones de la caja de mensajes (estilo de la MessageBoxA)
:0040136B 6860214000            push 00402160 <- empuja a la pila un puntero[6] al
encabezado de la caja del mensaje "No luck"

:00401370 6869214000            push 00402169 <- puntero a la cadena de texto que
ocupará el área cliente de  la ventana "No luck there, mate!"

:00401375 FF7508                push [ebp+08] <- empuja a la pila el valor del manejador
de la ventana padre, hWnd

:00401378 E8BD000000            Call 0040143A <- llama a la API MessageBoxA del
user32.dll de Windows.

:0040137D C3                    ret  <- fin del procedimiento
```

Además podemos decir que todas las líneas de código situadas entre el primer ret y el último son un *procedimiento o una función*, esto es, una unidad dentro del programa que realiza una tarea concreta, en este caso pintar una caja de mensaje. A los procedimientos se les *llama* desde un call y cuando terminan de ejecutarse *retornan* el control a la línea siguiente a su llamada. Si miramos un poco mas arriba vemos:

```
* Referenced by a CALL at Address:
|:00401245
```

Lo cual quiere decir que este procedimiento es llamado desde 401245. Buscando en Wdasm esa parte del código nos encontramos con:

```
* Possible StringData Ref from Data Obj ->"DLG_REGIS"
                        |
:00401213 6815214000            push 00402115
:00401218 FF35CA204000          push dword ptr [004020CA]

* Reference To: USER32.DialogBoxParamA, Ord:0000h
                        |
:0040121E E87D020000            Call 004014A0
:00401223 83F800                cmp eax, 00000000
:00401226 74BE                  je 004011E6
:00401228 688E214000            push 0040218E
:0040122D E84C010000            call 0040137E
:00401232 50                    push eax
:00401233 687E214000            push 0040217E
```

```
:00401238 E89B010000          call 004013D8
:0040123D 83C404              add esp, 00000004
:00401240 58                  pop eax
:00401241 cc 3BC3             cmp eax, ebx
:00401243 7407                je 0040124C
:00401245 E818010000          call 00401362
:0040124A EB9A                jmp 004011E6

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:00401243(C)


:0040124C E8FC000000          call 0040134D
:00401251 EB93                jmp 004011E6
```

Resaltado en rojo una parte del código que siempre debe suscitar nuestra atención, aunque recordad, crackear no consiste en ir buscando como locos un cmp o un je/jne, consiste en saber que hace el programa y como lo hace. Vemos que según sea la comparación de 0401241 se realiza el procedimiento 401245 (que acabamos de ver) o el de 0040134D. Veamos como es este:

```
* Referenced by a CALL at Address:
|:0040124C
|
:0040134D 6A30                push 00000030

* Possible StringData Ref from Data Obj ->"Good work!"
                        |
:0040134F 6829214000          push 00402129

* Possible StringData Ref from Data Obj ->"Great work, mate!"
                        |
:00401354 6834214000          push 00402134
:00401359 FF7508              push [ebp+08]

* Reference To: USER32.MessageBoxA, Ord:0000h
                        |
:0040135C E8D9000000          Call 0040143A
:00401361 C3                  ret
```

Es otro MessageBoxA que ya sabréis interpretar vosotros mismos. Saca otra caja de texto pero conteniendo un mensaje de éxito “Great work, mate!” Al parecer no hace otra cosa. En resumen, si introducimos un serial válido nos saca un mensaje diciendo que hemos acertado y nos manda a 004011E6 y si fallamos nos saca un mensaje diciendo que hemos fallado y nos manda también a 004011E6. Si fuera un *programa real* cabria esperar que en caso de acertar metiera nuestra información en alguna clave de registro o en algún fichero y la guardara ahí para recordar que nos hemos registrado correctamente. Pero esto es un crackme, no hace nada más que eso.

Búsqueda a lo retro

Ahora vamos a mostrar otra forma de llegar a lo mismo. Como sabemos que el programa lanza un mensaje de error cuando fallamos, podemos poner un Bpx en MessageBoxA con el Sice y así caer en una parte del código que posiblemente se ejecute justo después de realizar la comparación del serial. Para ello arrancamos el programa, ponemos datos al azar, ponemos el Bpx MessageBoxA y pulsamos ok. Los lectores más experimentados y también aquellos que hayan leído hasta aquí con más atención sabrán ya exactamente que línea va aparecer resalada en el Sice cuando este interrumpa la ejecución. Justamente la primera línea de la API MessageBoxA. Como sabemos que ha sido llamada como procedimiento después de empujar a la pila unos determinados valores, sabemos que devolverá la ejecución al programa después de uno varios rets, con lo que será suficiente con pulsar F12 (ejecuta hasta el siguiente ret) hasta que volvamos a caer en el programa principal, lo cual sabremos mirando la línea verde que separa la ventana de comandos de la ventana de código en el Sice, que debe decir crackme1! code. Después de pulsar una vez F12 llegamos a una parte de código que ya conocemos:

```
:00401378 E8BD000000        Call 0040143A <- aquí llama a MessageBoxA

XXXXXX                      push ebp <- primera línea de MessageBoxA. Su
número depende de la versión de Windows que estemos usando. Pulsamos F12

:0040137D C3                ret ← Venimos aquí. Pulsamos F10

:00401241 cc 3BC3           cmp eax, ebx
:00401243 7407              je 0040124C
:00401245 E818010000        call 00401362
:0040124A EB9A              jmp 004011E6    venimos aquí

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:00401243(C)

:0040124C E8FC000000        call 0040134D
:00401251 EB93              jmp 004011E6
```

Otra parte conocida. Como vemos se cae en la ejecución del programa justo después de realizada la comparación, por eso el nombre de *retro*. Podemos poner un Bpx en :00401241 y ver que es lo que se compara. Una vez paramos ahí hacemos ¿ eax y ¿ ebx y vemos 000054B3 y 000054B2. UHF… casi acertamos!
Pero… ¿Por qué se comparan estos dos números si ninguno de ellos es el que nosotros introdujimos? Imaginemos que eax valiera “18055” y ebx “18054”. Reconoceríamos nuestro serial y deduciríamos que compara el serial bueno (18055) con el nuestro (18054) y que como no son iguales saca el cartel de chico malo. Aunque no olvidemos una cosa, el ordenador

trabaja en base hexadecimal, no decimal, entonces el valor que veríamos en EBX seria 4686. Sin embargo vemos 54B2, luego nuestro serial ha sufrido una transformación antes de ser comparado. Es lo que se llama encriptación, o sea, ocultamiento. De este modo si ponemos que el serial es 54B3 seguiremos sin poder registrarnos, ya que su valor encriptado es 7FB.

Aún hay otra forma de llegar *via retro* hasta la zona del programa que nos interesa, pero esta es más propia de sistemas de protección por nag y será estudiada en el correspondiente capítulo ;-).

Por Predicción: Anticipándonos a la rutina de verificación.

En el capítulo 1 veíamos como BlackFenix hacía una clasificación de las formas de abordar los distintos sistemas de protección, que como veis estamos siguiendo. Es importante seguir un método, sobre todo al principio, así nos evitamos ir divagando de WDasm a Sice, de Sice a WDasm y de paso desgastar la tecla F10. En este capítulo se hablaba del sistema de *predicción* para llegar al núcleo de la protección. Cuando hacemos clic en register se nos abre una ventana donde metemos los datos. Por predicción podemos decir: Es un DialogBoxParam !! Ponemos un Bpx ahí y para. Efectivamente es un DialogBoxParam, aunque esto no tiene mucho merito, lo hemos visto ya en el listado muerto. No siempre se consigue ver cual es la API y ni siquiera se consigue siempre ver el listado muerto, por eso el método de predicción. En la práctica ponemos un Bpx en *todas* las API's que pensamos que pueden ser usadas por el programa *antes* de que se ejecute *eso* que nos interesa. En nuestro ejemplo ponemos un Bpx en DialogBoxParam, salta el Sice y podemos seguir traceando hacia delante hasta llegar, por ejemplo, al cartel de chico malo. En la primera pasada tal vez no veamos mucho pero tendremos la certeza de que hemos *pasado por encima* de la rutina de verificación del serial. Más adelante podremos meternos con F8 en los procedimientos (CALL's) y ver exactamente que hace cada uno.

Otro breakpoint que nos será de utilidad es el que ponemos en el API que toma el serial que nosotros hemos metido en la caja de diálogo. Traceando hacia delante desde aquí podremos ver que hace con nuestro serial hasta que lo compara con el bueno. Hay toda una lista de API's que hacen esto con ligeras variaciones. En este caso es GetDlgItemTextA. Vemos que para en:

```
:004012B5 6A0B                  push 0000000B
:004012B7 688E214000            push 0040218E

* Possible Reference to Dialog: DLG_REGIS, CONTROL_ID:03E8, ""
                                 |
:004012BC 68E8030000            push 000003E8
:004012C1 FF7508                push [ebp+08]

* Reference To: USER32.GetDlgItemTextA, Ord:0000h
                                 |
:004012C4 E807020000            Call 004014D0
:004012C9 83F801                cmp eax, 00000001

* Possible Reference to Dialog: DLG_REGIS, CONTROL_ID:03EB, "Cancel"
                                 |
:004012CC C74510EB030000        mov [ebp+10], 000003EB
:004012D3 72CC                  jb 004012A1
:004012D5 6A0B                  push 0000000B
:004012D7 687E214000            push 0040217E

* Possible Reference to Dialog: DLG_REGIS, CONTROL_ID:03E9, ""
                                 |
:004012DC 68E9030000            push 000003E9
:004012E1 FF7508                push [ebp+08]

* Reference To: USER32.GetDlgItemTextA, Ord:0000h
                                 |
:004012E4 E8E7010000            Call 004014D0
:004012E9 B801000000            mov eax, 00000001
:004012EE EB07                  jmp 004012F7
```

Echemos un vistazo a las características de este API para saber que se está empujando a la pila:

La función GetDlgItemText recupera el título o texto asociado con un control en una caja de diálogo.

```
UINT GetDlgItemText (
  HWND hDlg,                    // el manipulador de la caja de diálogo
  int nIDDlgItem,               // el identificador de control
  LPTSTR lpString,              // la dirección del buffer para el texto
  int nMaxCount                 // el tamaño máximo de string

);
```

Returns
Si la función tiene éxito, el valor de retorno especifica el número de caracteres copiados al buffer, no incluido el carácter nulo de terminación.

Si la función falla, el valor de retorno es cero.

Vemos que hay dos llamadas consecutivas a GetDlgItemTextA. La primera es la toma del texto de la caja name.

```
:004012B5 6A0B                  push 0000000B <- 10d (Bh) es el
tamaño máximo de la cadena que toma. Podemos escribir más caracteres,
pero solo tomará 10.
:004012B7 688E214000            push 0040218E <- Aquí guarda
nuestro "name"

:004012BC 68E8030000            push 000003E8 <- identificador del
control (caja de texto)
:004012C1 FF7508                push [ebp+08] <- manipulador
(handle) de la caja de dialogo.

:004012C4 E807020000            Call 004014D0 <- llamada a
GetDlgItemTextA
```

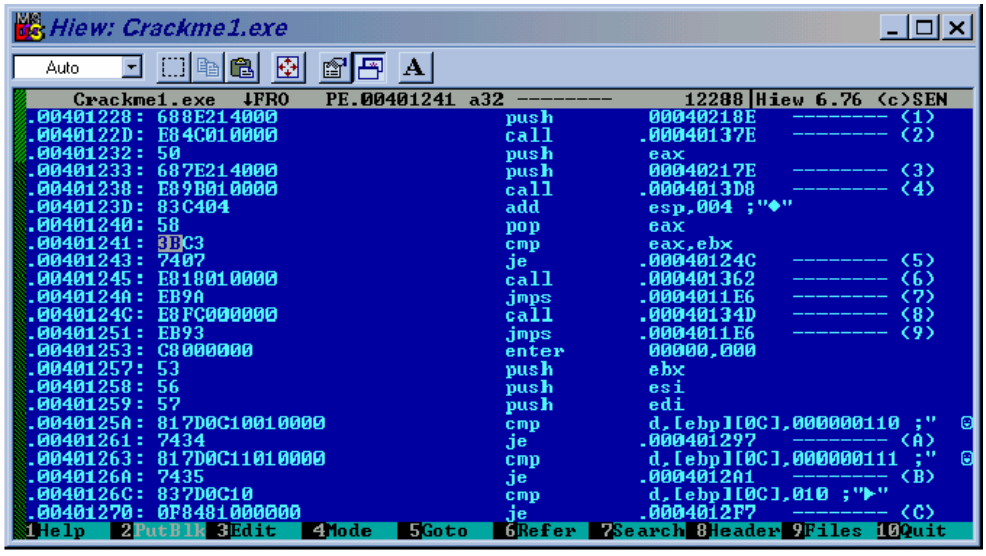
La segunda la toma de la caja serial:

```
:004012D5 6A0B                  push 0000000B
:004012D7 687E214000            push 0040217E <-Aquí guarda nuestro
serial
:004012DC 68E9030000            push 000003E9
:004012E1 FF7508                push [ebp+08]
:004012E4 E8E7010000            Call 004014D0
```

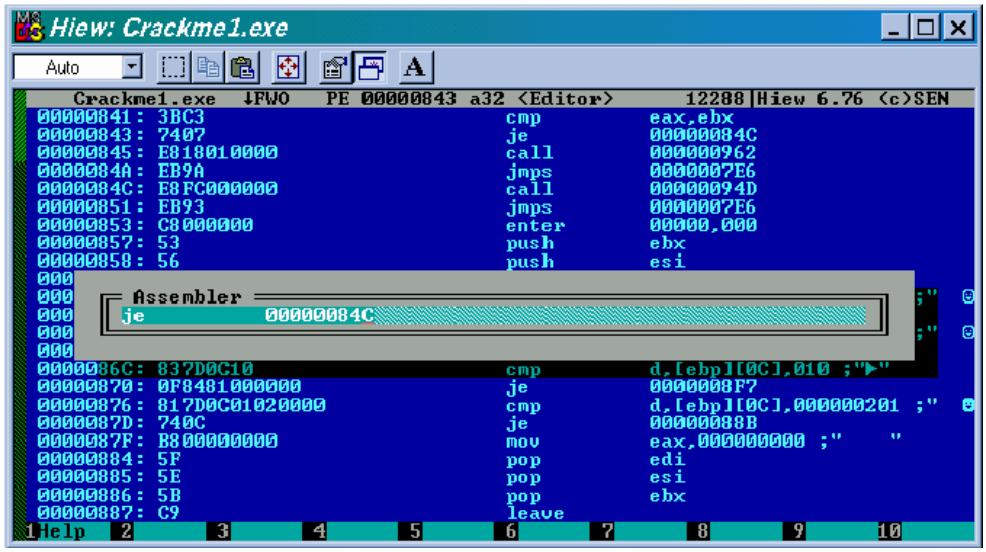
Todas estas aproximaciones que hemos mostrado sirven para aproximarnos a la zona caliente, esa en la que se decide si vamos a estar registrados o no. Una vez identificada tenemos distintas formas de eliminar la protección, lo cual depende bastante del tipo de protección de que se trate y de cómo esté construida. De un modo general y aplicando un poco más especialmente al caso de los seriales y, en concreto, a nuestro ejemplo, tenemos las siguientes formas.

El inevitable parcheo

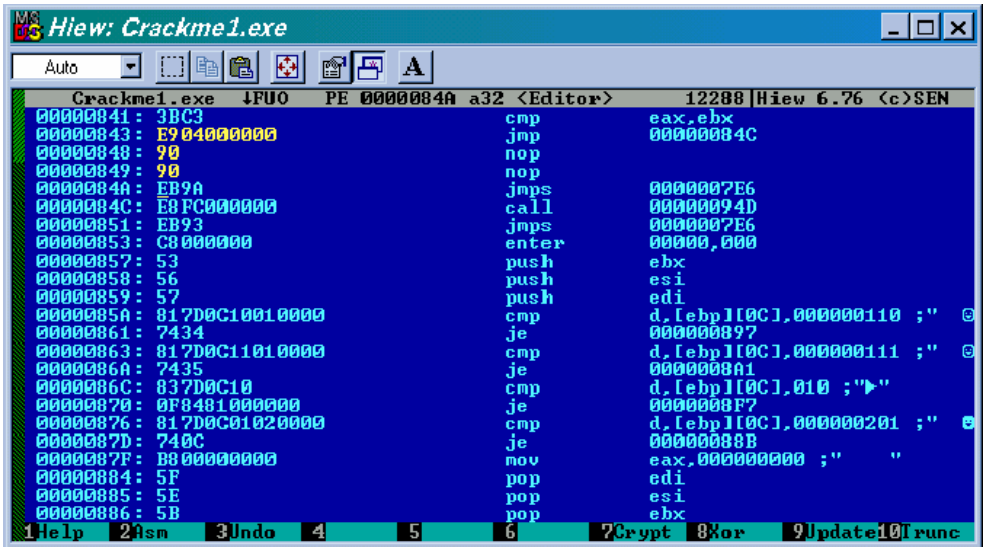
Parchear un programa significa cambiar unos bytes por otros. Para ello se necesita identificar el offset de los bytes que queremos cambiar, localizarlos mediante el offset en un editor hexadecimal y cambiarlos. Si editamos un archivo exe lo que vamos a ver son los valores ASC de los opcodes en hexadecimal. Por lo tanto, lo que necesitamos es saber el valor en hexadecimal de la instrucción que vamos a cambiar y de la instrucción por la que queremos cambiarla, localizarla por medio del offset conocido y sustituirla. Esta es la explicación tradicional y la primera que leí yo al respecto! Pero para evitar procesos migrañosos u otros de peores consecuencias mi consejo es usar Hiew. Con Hiew podemos localizar el byte que nos interesa sin conocer su offset y parchearlo sin conocer el opcode de la instrucción ASM. Antes de nada haz una copia del ejecutable que vas a parchear. Abre Hiew, pulsa Alt+F2, selecciona la unidad, luego elige el directorio y el programa. Pulsa F4 y selecciona decode. Pulsa F5 y la dirección RVA (virtual adress) que quieres cambiar precedida de un punto .401241. Aquí es donde vimos que se hacía la comparación del serial encriptado.



En la siguiente instrucción, JE 40124C, se realiza el salto solo cuando hemos puesto un serial correcto. Una posibilidad seria cambiar JE por JNE, o lo que es lo mismo, 74 por 75. Entonces el programa dará por buenos todos los números *menos* el realmente válido, lo cual aumenta considerablemente nuestras posibilidades. Para esto hacemos F3 (edit), F2 (asm) y donde pone JE ponemos JNE:



pulsamos F9 (update) y F10 (quit) y parcheado. Como podéis ver en la imagen, a la izquierda, ahora aparecen los offsets en lugar de la dirección en memoria en las instrucciones (RVA). Esto sucede cuando pulsamo F3, que puede servir también para esto. Posiblemente tú hayas pensado en otras formas de parchear. Otra más que se utiliza mucho es hacer un JMP para que salte siempre, sea cierta o no la comparación, aunque hay un problemilla, veamos:



Como podéis ver la instrucción original JE 40124C se codifica como 7407, que ocupa 2 bytes. La nueva instrucción, JMP 40124C, se codifica como E904000000, que ocupa 5 bytes con lo que invade los 3 primeros bytes de la siguiente, que es el CALL que muestra el cartel de chico malo. Como a nosotros ese cartel no nos interesa en ningún caso, terminamos de quitarlo añadiendo 2 instrucciones nop para que no se produzca error por desplazamiento del marco de lectura (como en las Talasemias). Seguramente habréis pensado otro buen montón de formas de parchear este programa. Aparte del clásico 74 por 75 y viceversa y el nopeo del CALL maldito, como dicen los castizos, no existen más reglas predefinidas, sino que es la compresión del sistema de protección y el análisis de su código lo que nos hace decidir el parcheo más conveniente.

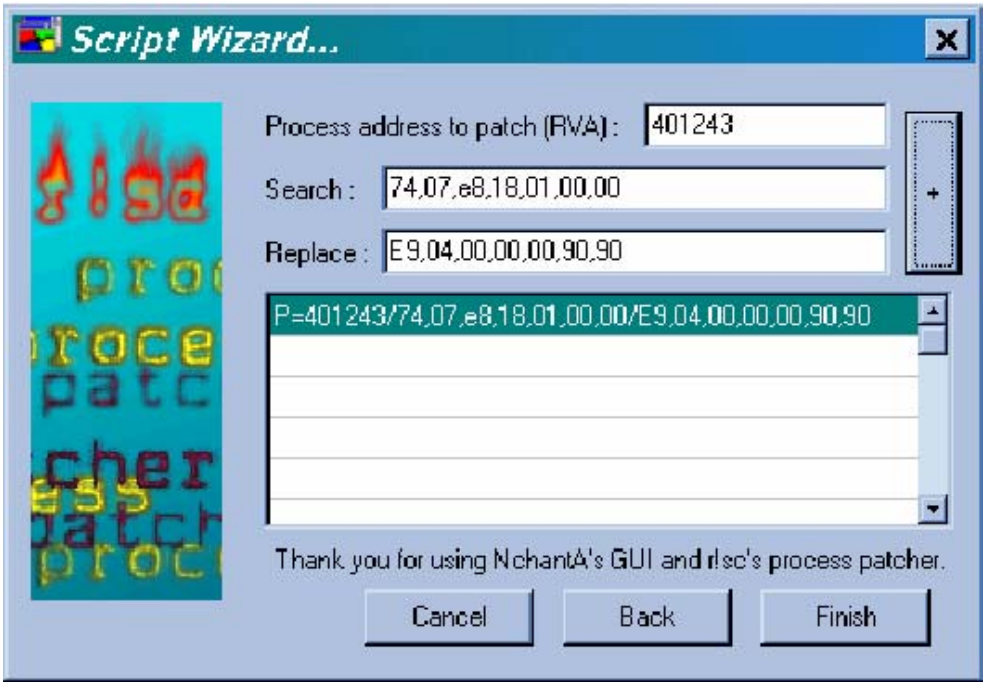
El crack distribuible

Otro aspecto de este tema, quizás el más interesante por cuestiones obvias, es la creación de *patchers*, esto es, pequeños programitas que toman un ejecutable original y parchean esos bytes *defectuosos* hasta dejarlo de nuestro gusto. Tenemos una variedad de programas que hacen esto, dependiendo nuestra elección del aspecto gráfico del patcher, el tamaño que ocupa, la cantidad de bytes que puede parchear y nuestras preferencias personales. Uno que a mi me gusta bastante es Cogen, del grupo Egoiste, que se puede bajar de coders domain. Genera parcheadotes en Turbo Assembler a partid de un fichero asm que contiene el código fuente que nosotros podemos modificar personalizándolo. Simplemente deberemos tener el programa original sin parchear y otro parcheado con el Hiel, pulsar el botón Generate y listo, un pequeño parcheador de tan solo unas kb's que podemos mandar cómodamente por e-mail a un amigo para que reviva esa aplicación caducada…

Además de parchear físicamente el archivo ejecutable, podemos parchear el programa una vez que se ha cargado en memoria. Esto es lo que hacen los

loaders. Permiten obtener el efecto deseado sin modificar el archivo original, con la condición de que lo arranquemos siempre desde el loador. Su utilidad es especialmente relevante en el caso de ejecutables comprimidos. Aunque este tema nos ocupará durante todo un volumen, digamos ahora que el ejecutable comprimido no puede ser desensamblado con WDasm ni tampoco permite parchear lo que vemos con el Sice. Su ejecución comienza por una rutina que desempaqueta el resto del ejecutable y lo carga en memoria, donde realmente se encuentra el programa en sí tal cual era antes de que se le aplicara un empacador. En este tipo de archivos no funcionará un patcher, pero un loador es posible que sí lo haga. Los loaders se diferencian en cuanto al modo que tienen de actuar y también de acuerdo a características como las que describíamos para los patchers. Un programa muy usado es el Risc Process Patcher, otra joya escrita en TurboAsm que nos permite ver el código fuente y modificarlo a nuestro gusto y que además trae una completa interfaz gráfica con la que podemos crear nuestro loador en tres pasos.

Apretamos  y ya estamos creando un loador. En la primera ventana ponemos el nombre del programa y el nuestro, solo con fines informativos (al FBI puede interesarle ;-). En la siguiente ponemos el archivo que queremos parchear y el nombre que le daremos al loador. La última es la más importante.



Ponemos la dirección virtual (RVA) donde esta la primera instrucción que queremos cambiar. Escribimos en search el código de la instrucción ASM original separando cada byte por una coma y lo mismo para la instrucción con la queremos sustituir. Aquí si que tenemos que conocer los opcodes, pero eso lo podemos mirar con Hiew. Pulsando Finish y luego Save and Build script  tenemos hecho el loador. Si lo ejecutamos lo que veremos es que se

comporta igual que si estuviera físicamente parcheado.

Obtener un serial

La mejor forma de crackear un programa siempre es introducir un serial válido. El programa no es modificado y todas las rutinas de comprobación y de registro verifican nuestra condición de usuarios autorizados. No solo modificamos aquellas rutinas que "engañan" al programa haciéndolo creer que lo estamos, lo cual demasiadas veces puede conducir a errores imprevistos. Obtener un serial mediante ingeniería inversa supone una considerable condición de esfuerzo adicional ya que significa que tenemos un conocimiento mucho más exacto de la protección del programa que el requerido para un simple parche. También significa que la protección del programa es más débil ya que hay que desmontar todas las rutinas de encriptación partiendo de un listado de ASM en el mejor de los casos. Sin embargo nuestro amigo crackme es un ejemplo sencillo del método a seguir. Si vemos el trozo de código de la página 5, que es donde se encuentra el meollo, veremos el DlgBoxParam seguido de unas cuantas líneas de código hasta llegar a la comparación decisiva. Sabemos que ahí se realiza la encriptación de nuestro serial y del serial bueno y después se comparan. Como no vemos en ese listado las instrucciones de encriptación, forzosamente han de estar contenidas en alguno de los CALL. Si traceamos un poco con el Sice en seguida veremos como se hace esto:

```
:0040121E E87D020000        Call 004014A0       <- llama a DlgBoxParam
:00401223 83F800            cmp eax, 00000000   <- verifica el valor de retorno de la
API
:00401226 74BE              je 004011E6         <- si ese valor es cero (fallo de la
función) nos manda a otra parte
:00401228 688E214000        push 0040218E       <- Empuja nuestro name a la pila
:0040122D E84C010000        call 0040137E       <- EAX=54B3; se ha generado
un número a partir de nuestro name
:00401232 50                push eax
:00401233 687E214000        push 0040217E       <- Empuja nuestro serial
:00401238 E89B010000        call 004013D8       <- EBX=54B2
:0040123D 83C404            add esp, 00000004   <- sitúa el siguiente valor de la
pila en ser leído
:00401240 58                pop eax             <-EAX=54B3
:00401241 cc 3BC3           cmp eax, ebx        <- Compara nuestro serial
encriptado (54B2) con el resultado de manipular nuestro name (54B3)
:00401243 7407              je 0040124C         <- Chico Bueno
:00401245 E818010000        call 00401362       <- Chico Malo
:0040124A EB9A              jmp 004011E6
```

Asi pues, parece que el meollo se cocina en esos dos CALL's. Veámoslo de más cerca:

```
* Referenced by a CALL at Address:
|:0040122D
|
:0040137E 8B742404                mov esi, dword ptr [esp+04] <- ESI
es un puntero a name
:00401382 56                      push esi  <- empuja ESI a la pila

* Referenced by a (U)nconditional or (C)onditional Jump at Addresses:
|:00401392(U), :0040139A(U)
|
:00401383 8A06                    mov al, byte ptr [esi]<- al
contiene el carácter de name apuntado por ESI, inicialmente el 1°
:00401385 84C0                    test al, al <- es cero?[15]
:00401387 7413                    je 0040139C <- si? fuera del bucle
:00401389 3C41                    cmp al, 41  <- 41h es 65d y la A en
código ASC
:0040138B 721F                    jb 004013AC <- al es una carácter
previo a A? Vamos a un MessageBoxA de chico malo idéntico al primero
que vimos (redundancia de código)
:0040138D 3C5A                    cmp al, 5A   <- 5Ah es 90d y Z ASC.
:0040138F 7303                    jnb 00401394 <- al es un carácter
posterior a Z? Salta aun CALL en 401394.
:00401391 46                      inc esi      <- ESI aumenta en 1,
luego byte ptr [ESI] apunta al siguiente carácter de la cadena.

:00401392 EBEF                    jmp 00401383 <- Va al principio del
bucle.

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:0040138F(C)
:00401394 E839000000              call 004013D2 <- resta 20h (32d) a
al. Esto hace que si al=61h (97d) "a" menos 20h = 51h = A - > Pasa de
minúsculas a mayúsculas

:00401399 46                      inc esi
:0040139A EBE7                    jmp 00401383  <- Va al principio
del bucle.

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:00401387(C)
|
:0040139C 5E                      pop esi       <- Name en mayúsculas
:0040139D E820000000              call 004013C2 <- Más CALL`s…

    * Referenced by a CALL at Address:
    |:0040139D
    |
    :004013C2 33FF                    xor edi, edi <- Borra EDI
    :004013C4 33DB                    xor ebx, ebx <- Borra EBX

    * Referenced by a (U)nconditional or (C)onditional Jump at
    Address:
    |:004013CF(U)
    |
    :004013C6 8A1E                    mov bl, byte ptr [esi] <-
    Pasa a bl el primer carácter en mayúsculas de name
    :004013C8 84DB                    test bl, bl
    :004013CA 7405                    je 004013D1 <- Final de
    cadena ? Manda a un RET desde donde se vuelve a 00401232
    :004013CC 03FB                    add edi, ebx <- EDI es la
    sumatoria del valor ASC en hexadecimal de todos los caracteres
    de name (recordemos que como máximo acepta 10 caracteres)
    :004013CE 46                      inc esi      <- Apunta al
    siguiente carácter
    :004013CF EBF5                    jmp 004013C6 <- Vuelve al
    inicio del bucle
```

```
:004013A2 81F778560000            xor edi, 00005678 <- Hace un XOR de
EDI con 5678
:004013A8 8BC7                    mov eax, edi      <- Lo pasa a EAX
:004013AA EB15                    jmp 004013C1      <- Salta al RET

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:0040138B(C)
|
:004013AC 5E                      pop esi
:004013AD 6A30                    push 00000030

* Possible StringData Ref from Data Obj ->"No luck!"
                                   |
:004013AF 6860214000              push 00402160

* Possible StringData Ref from Data Obj ->"No luck there, mate!"
                                   |
:004013B4 6869214000              push 00402169


:004013B9 FF7508                    push [ebp+08]

* Reference To: USER32.MessageBoxA, Ord:0000h
                                    |
:004013BC E879000000                Call 0040143A

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:004013AA(U)
|
:004013C1 C3                        ret <- Vuelve a la línea principal
```

Mensaje de chico malo si al < 41h

Ya veis que fácilmente hemos desenmascarado la primera de las rutinas de encriptación. Inicialmente hace la sumatoria del valor ASC de los 10 primeros caracteres es inferior a 40h nos saca el cartel de chico malo *dos veces*! Si el valor es mayor de 5ª le restara 20h, lo cual tiene la virtud de convertir las minúsculas a mayúsculas aunque también modifica el valor de otros caracteres no alfabéticos que introduzcamos. Luego toma la sumatoria de EDI y hace un XOR con 5678. Esa es la cifra que finalmente compara con la que calcula en el siguiente CALL.

Para ahorrarnos un poco de código, que ya llevamos bastante, resumiremos de palabra lo que hace el segundo CALL, el que encripta el serial, y aquellos que lo deseen que lo comprueben por si mismos. Este CALL convierte el valor literal de la cifra que introducimos como serial en un valor numérico. Después hace un XOR con 1234 y pasa el valor a EBX, que es el registro que se compara con EAX para decidir si hemos dado un serial válido o no.

Llegados a este punto nada nos impide hacernos nuestro propio serial. Siguiendo con el ejemplo inicial, el correspondiente a mR_gANDALF es 18055. Igual que lo hacemos para este nombre podemos hacerlo para cualquier otro. Pongamos manos a la obra.

Keygen, el mejor crack

Cuando vemos un keygen sabemos que el cracker que lo ha hecho conoce la

protección del programa al menos tan bien como el programador que lo ha hecho y posiblemente mejor. Me encanta leer los *.nfo y los *.diz que vienen con los keygen de esos grandes crackers. Son bonitos programas (generalmente en ensamblador) que generan las claves a partir de los datos que necesitan para calcularlos. Existen herramientas verdaderamente interesantes para realizar los keygens. TMGRipper tracea desde el punto de inicio de la rutina encriptadora siguiendo todas las ramificaciones y los CALL's hasta que llega a un endpoint que le hemos marcado o encuentra una salida de texto a la pantalla, como seria un cartel de chico malo. KeGenMaker es un programa en TurboAsm de crackers polacos que nos permite añadir la rutina de encriptado al esqueleto de un programa ASM, que es la interfaz del KeyGen. Para MASM hay varias utilidades que hacen esto mismo, como Muad´dib's. Se pueden hacer KeyGens en ASM con poco esfuerzo utilizando estas bonitas herramientas, pero eso es materia de otro capítulo ;-) Sin embargo, vamos a realizar nuestro primer keygen en Visual Basik 6 para simplificar bastante la tarea. Como en cualquier lenguaje, lo primero que tenemos que hacer antes de empezar a teclear código es tener una idea exacta de lo que va a hacer el programa. En nuestro caso una función que calcule el serial válido en base a un nombre de 10 caracteres o menos dado. Dicho serial ha de cumplir la siguiente condición:

$$\text{Xor serial, 1234} = \text{Xor } (\sum \text{ASC name}), 5678$$

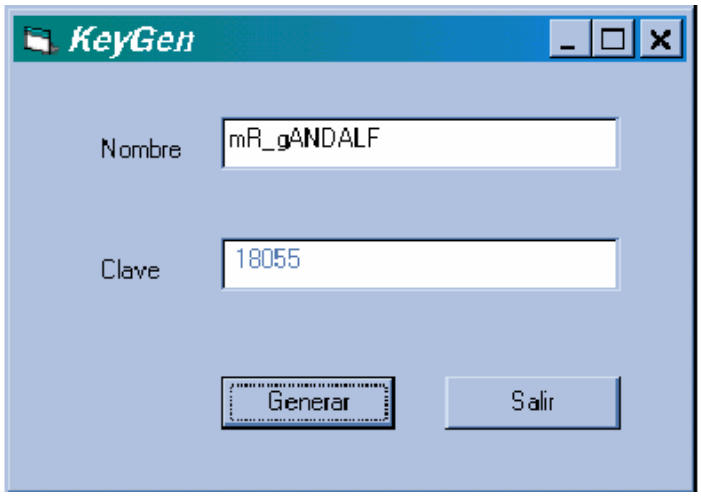
Sabiendo que Xor es la función inversa de sí misma.:

$$\text{Serial} = \text{Xor } (\text{Xor } (\sum \text{ASC name}), 5678), 1234$$

Entonces calcularemos el serial elegido en 3 sencillos pasos:

1. Calcular la sumatoria del valor ASC de cada letra de name
2. Hacer un Xor entre el número anterior y 5678
3. Hacer un Xor entre el número anterior y 1234

Estas son las funciones de VB que debemos escribir, además de una sencilla interfaz como la de la imagen.



La caja de texto correspondiente a clave está deshabilitada y solo se llena con un serial calculado en base al contenido de Nombre cuando clickeamos Generar. Debemos escribir las siguientes funciones:

1. Cálculo de la sumatoria del valor ASC de las letras de Nombre: Función SUMATEXTO().
2. Cálculo de la equivalencia decimal a binario y viceversa: binary() y Base_10()
3. Cálculo de la función Xor, que sólo se puede hacer a partir de números binarios y que VB solo emplea como operador lógico de relación y no como operador binario, a diferencia de ASM, por ejemplo, función F_Xor().

El cuerpo principal del programa quedaría como sigue:

```
Option Explicit
Private Sub Command1_Click()
Dim I As Byte, suma_binaria As String, suma_binaria_filtro1 As String, filtro1
As String, filtro2 As String

Suma_binaria = binary (Str (SUMATEXTO(Text1)))   ← Calcula sumatorio
                                                   ASC name y lo pasa a
                                                   binario para hacer un xor
Filtro = binary (22136) ←  -22136d = 5678h

Suma_binaria_filtro1 = F_Xor (suma_binaria, filtro1) ← Xor de sumatorio
                                                       ASC name y 5678, equivale
                                                       a EAX de 00401241 cmp
                                                       eax, ebx
Filtro2 = binary (4660) ← -4600d = 1234h

If not text1 = "" Then text2 = Str(base_10(F_Xor(suma_binaria_filtro1,
filtro2))) ← Xor del nombre encriptado y 1234h, valor final del serial

End Sub
```

Y eso es todo por este capítulo, el que cierra el primer volumen.

mR gANDALF