Introducción al cracking

2014

1/13/2014
HackingMexico
Alejandro Torres Ramírez - TorresCrack

Twitter: https://twitter.com/TorresCrack248
Facebook: https://www.facebook.com/yo.torrescrack

e-mail: tora_248@hotmail.com

Vamos a comenzar, dentro de las primeras que veremos en el curso será aprender a configurar nuestras herramientas, explicando a detalle cada paso que daremos.

En este caso será una introducción al cracking para programas en Windows de 32 bits, esta introducción es válida para sistemas operativos Windows de x86.

Los procesadores X86 de Intel y compatibles tienen 4 niveles de privilegio:

Ring 0: mayor privilegio (nivel kernel), sobre este se ejecuta el sistema operativo

Ring 1

Ring 2

Ring 3: menor privilegio (nivel usuario)

Win32 solamente soporta ring 0 y ring 3, pero nosotros trabajaremos en nivel de usuario (ring 3) por eso de mayor compatibilidad y además que debemos comenzar por un nivel muy basico, existen algunas protecciones que se ejecutan en ring 0 y son muy fuertes, pero causan muchos problemas.

Cuando trabajamos con ring 3 estaremos invocando a las API's de Windows (de las que les explicare más adelante) y estas lo que hacen es entrara a nivel kernel (ring 0) a ejecutarse y después retornar a nivel usuario, de esta forma nos aísla del ring 0.

Conociendo y configurando nuestras herramientas:

OllyDebugger: Ahora les platicare acerca del debugger que usaremos, para los que no lo conozcan les platicare de él; Olly debugger es un debugger para programas en Windows de 32 bits y este es para trabajar en ring 3.

Winhex: Es el editor hexadecimal que quizas usaremos.

RDG Packer Detector: RDG Packer Detector es un detector de packers,Cryptors,Compiladores, Packers Scrambler,Joiners,Installers.

PID: es un detector de packers,Cryptors,Compiladores, Packers Scrambler,Joiners,Installers.

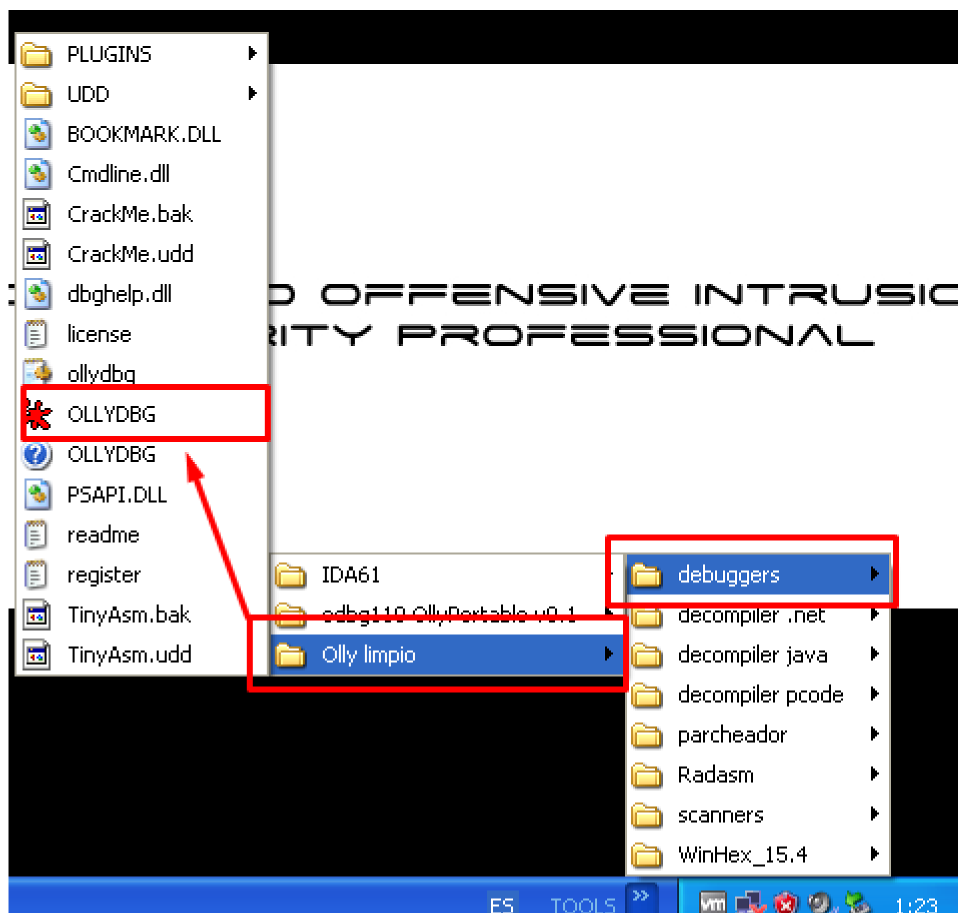
Radasm: hablemos de nuestro IDE Radasm, les cuento, Radasm es un entorno integrado de desarrollo que nos permite programar en diferentes lenguajes pero en este caso usaremos únicamente MASM.

Topo: Es una herramienta que nos ayuda a crear secciones dentro de los binarios o también buscar huecos vacíos para injertar código.

Dup2: Es una herramienta que nos ayuda a la creación de parches y loaders.

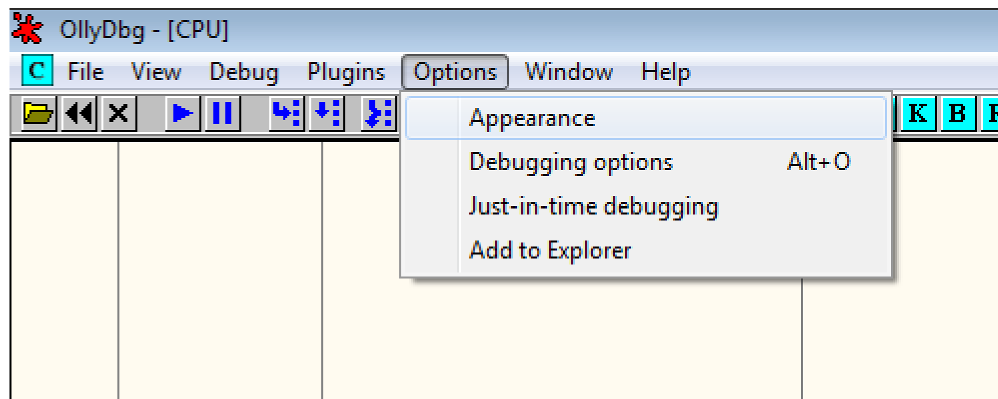
Después de una breve explicación de cada una de las herramientas, pasemos a configurar algunas de ellas:

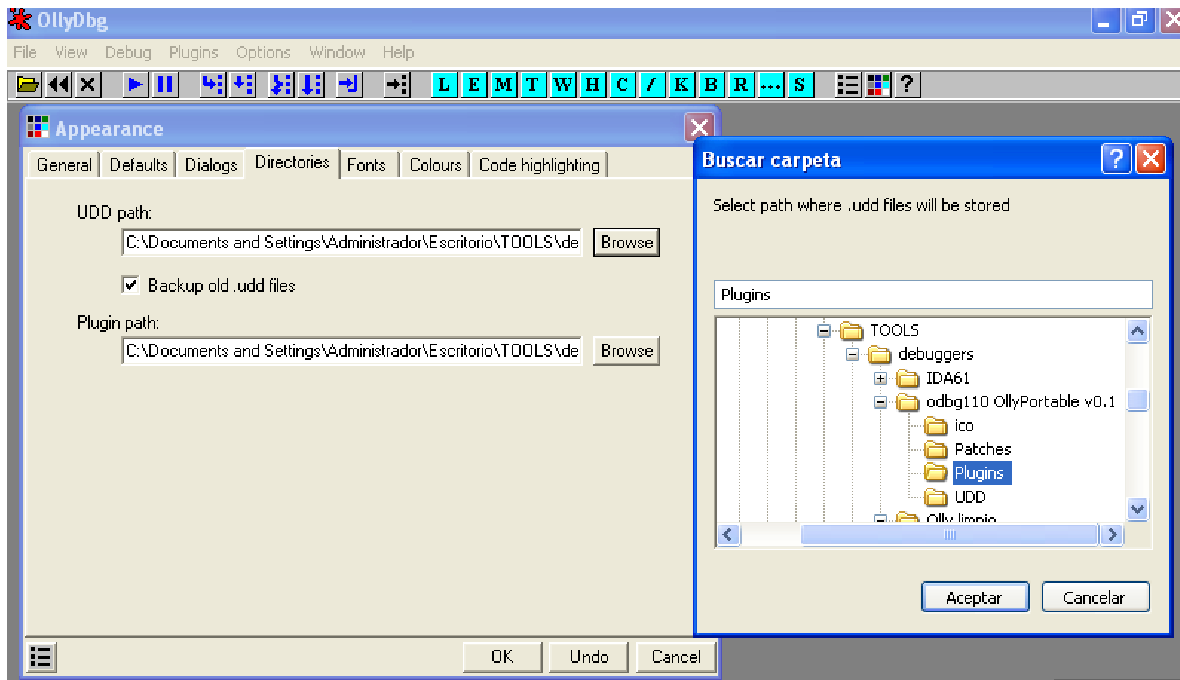
En la barra de tareas tenemos algunas de ellas:



Abrimos nuestro olly debugger limpio y pasaremos a configurar las rutas para seleccionar donde estarán las rutas donde se encuentran nuestros Plugins y donde se guardan los UDD que son los backup de cada binario que hemos analizado, cada cambio que hagamos en un ejecutable se guardara un backup con extensión .udd

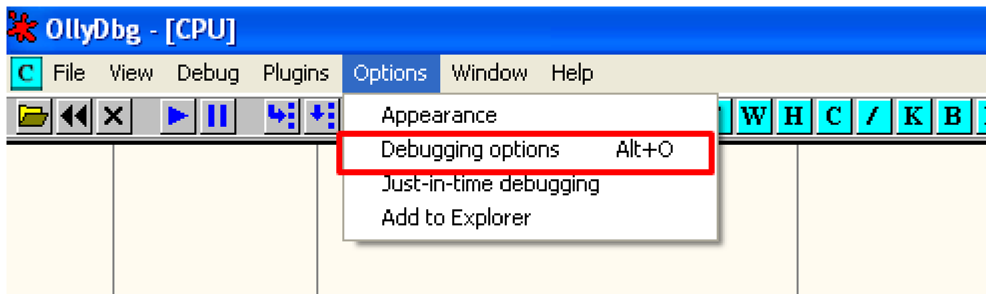
Ya una vez abierto en el menú se, entramos en “options” y seleccionamos “appearance”, lo cual nos mostrara una ventana para fijar las rutas:

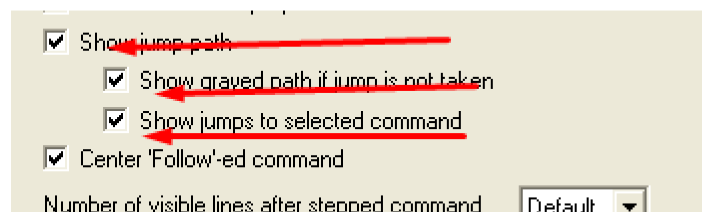




Nos pedirá que reiniciemos para guardar los cambios, le decimos que si, ahora vamos

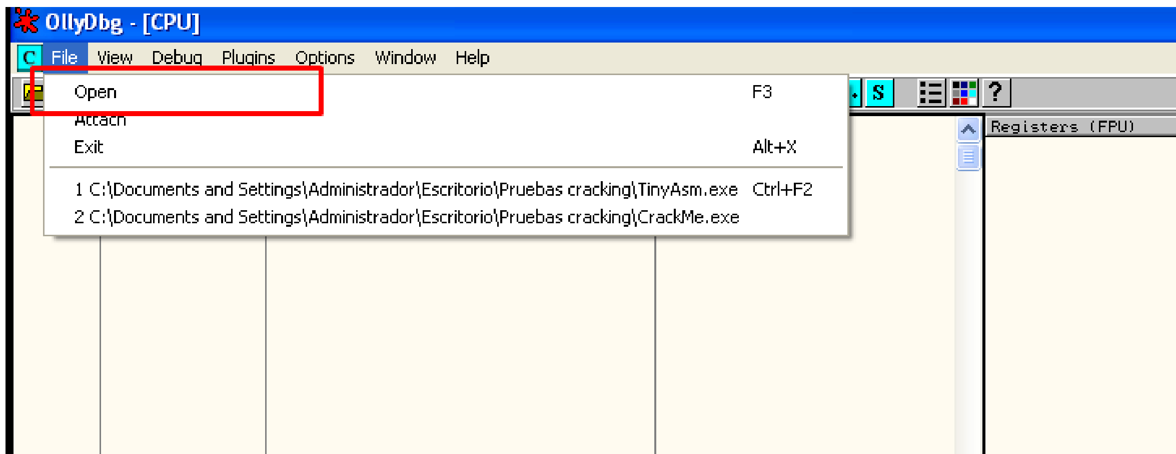
Ahora en “options” seleccionamos “debugging options”

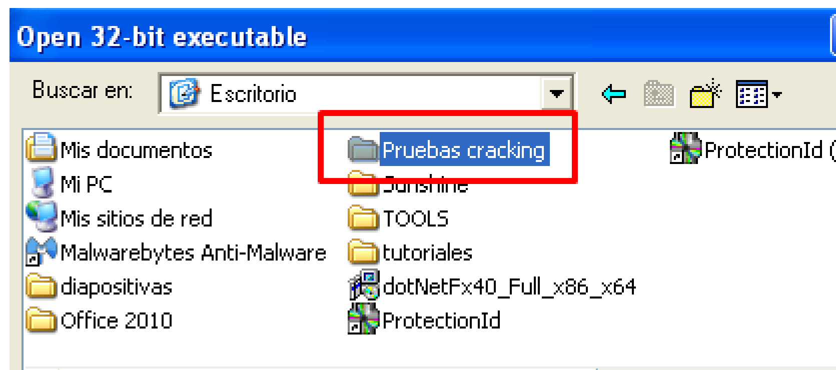


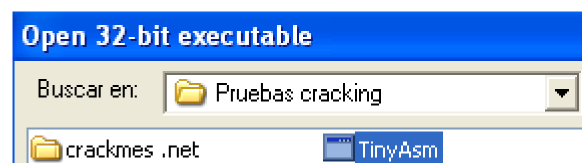


Estos cambios en la configuración se usan para facilitarnos el trabajo cuando analicemos un binario, ya que estos lo que harán será mostrarnos con una pequeña flecha hacia donde se dirigen los saltos condicionales de los cuales hablaremos más adelante y se los explicare.

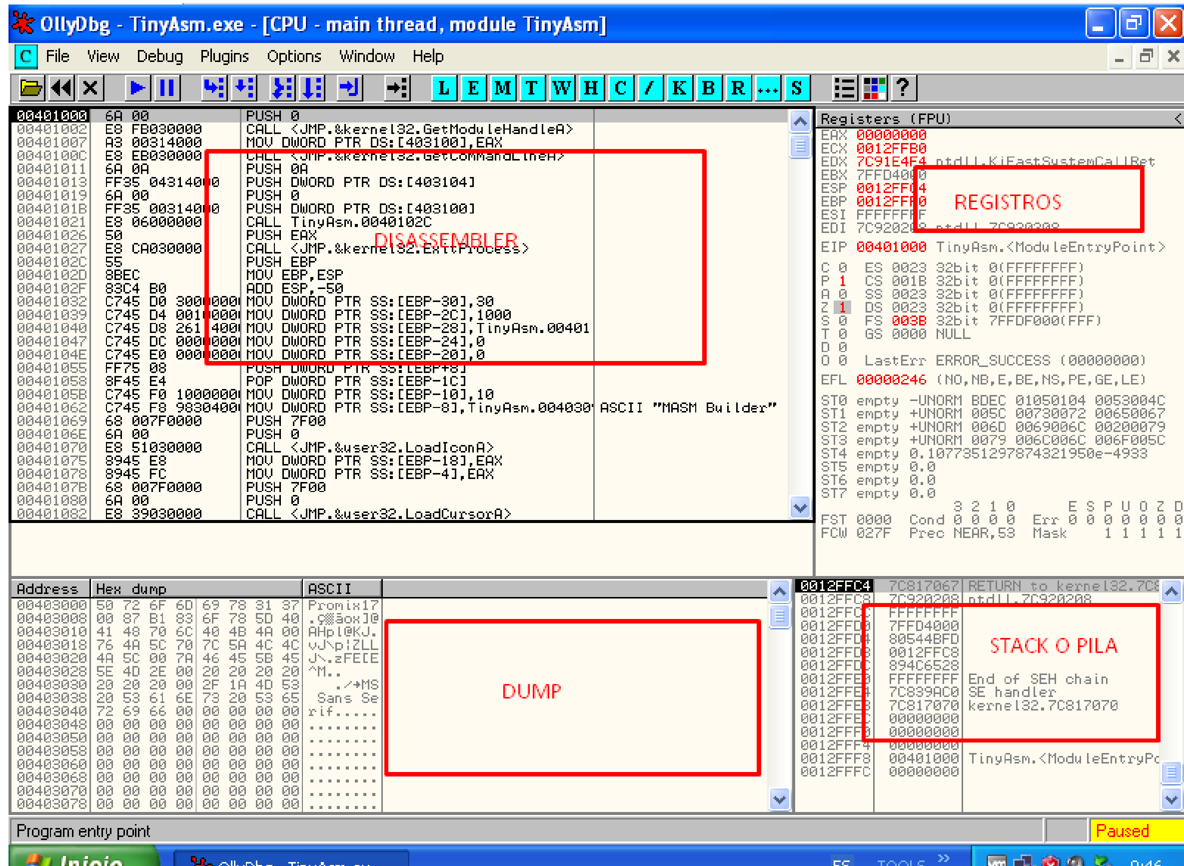
Ahora pasemos a lo siguiente y es abrir un ejecutable dentro de nuestro olly debugger en el menu "File" seleccionamos "open" y escogemos el binario que queremos ver empezar a analizar y seleccionamos en la carpeta del escritorio, la carpeta pruebas cracking seleccionamos el ejecutable "TinyAsm":



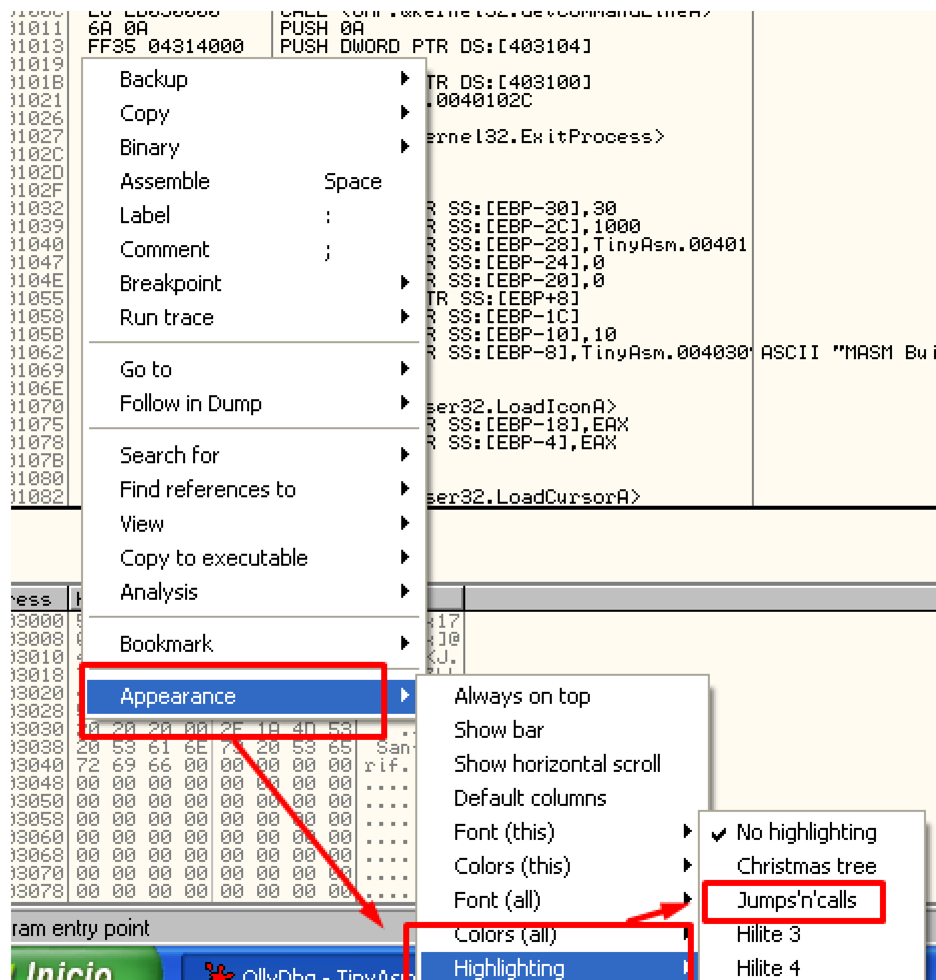




Al abrirlo dentro de nuestro debugger se vera algo asi, lo cual en la imagen vienen los nombres de cada una de de las partes que componen nuestro debugger:



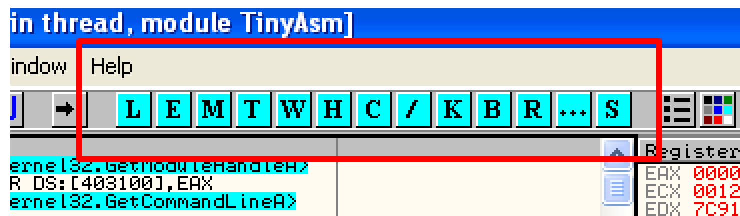
Damos un click en el disassembler y hacemos click derecho y seleccionamos “appearance” en el submenu damos en “Highlighting” despues en “jumps’n’calls”



Como verán al hacer esto se han pintado de colores para poder distinguir los saltos condicionales e incondicionales y algunas llamadas “calls” que pueden ser llamadas a un procedimiento o a alguna API de Windows.

Ahora vamos a explicar algunas partes esenciales:

Ahora vamos a explicar algunos botones de nuestro debugger para que tengamos un mayor control:



L: log data, muestra un detalle de lo que va haciendo el Olly (cuando arranca el programa, cuando genera un error, etc.)

E: executable modules, muestra todos los módulos que utiliza el programa debuggeado, el propio exe, las librerías que carga, etc.

M: memory map, como su nombre lo indica nos muestra un mapa de la memoria donde está nuestro programa, las dll que utiliza, etc

T: threads, nos muestra los hilos de ejecución que utiliza nuestro proceso .

W: windows, nos muestra las ventanas que tiene abiertas el programa.

H: handles, son los manejadores que utiliza nuestro programa (también lo vamos a ver bien más adelante)

C: cpu, la pantalla principal del Olly

/: patches, muestra los parches que se aplicaron al programa

K: call stack of main thread, muestra los distintos calls a los que vamos entrando

B: breakpoints, nos muestra los distintos breakpoints que hemos puesto en nuestro programa (lo que hacen es interrumpir la ejecución y darle el control al debugger)

R: references, nos muestra las referencias cuando realizamos alguna búsqueda

Ahora pasemos a explicar un poco de lenguaje ensamblador, no a fondo solo unas pocas instrucciones de ensamblador con las cuales nos encontraremos:

Assembler:

Registros del procesador:

Los registros los podemos ver como espacios físicos que residen dentro del procesador y se emplean para controlar instrucciones en ejecución, manejar direccionamientos de memoria y proporcionar capacidades aritméticas y lógicas. Siempre que hablemos de registros vamos a hacer referencia a los registros de 32 bits

Registro generales:

EAX: registro acumulador, es utilizado para obtener el valor de retorno de las API's, CreateFileA.

EBX: registro base, se suele utilizar para direccionar el acceso a datos situados en la memoria. También como el registro eax lo podemos dividir en BX, BH y BL.

ECX: registro contador, se utiliza como contador en determinadas instrucciones. También podemos usar CX, CH y CL.

EDX: registro de datos, además de su uso general. También se lo utiliza en operaciones de Entrada/Salida. Podemos utilizar EDX, DX, DH y DL.

Registros de puntero:

ESP: es un registro que apunta a la dirección del último valor introducido en la pila, o sea del primero que podríamos sacar. Cuando ingresamos o sacamos valores del stack el SO lo actualiza automáticamente para que siempre apunte al último valor. De todas formas podemos modificarlo desde nuestro código, ya veremos cómo. Pueden utilizarse los 16 bits inferiores con SP.

EIP: este registro apunta a la dirección de la próxima instrucción a ejecutarse y se va modificando automáticamente según se va ejecutando el programa.

Registros de base:

EBP: se utiliza para direccionar el acceso a datos situados en la pila y también para uso general. Pueden utilizarse los 16 bits inferiores con BP.
Registros de índice:
ESI y EDI: estos registros se utilizan para acceder a posiciones de memoria, por ejemplo cuando queremos trasferir datos de un lugar a otro o cuando queremos comparar dos bloques de memoria contigua. ESI actúa como puntero al origen (source) y EDI como puntero al destino (destination).

Podemos acceder a los bytes inferiores con SI y DI.

Los flags:

Las banderas también son un registro de 32 bits, donde cada uno de estos bits tiene un significado propio, que generalmente son modificados por las operaciones que realizamos en el código, y los cuales se los utiliza para tomar decisiones en base a las mismas: comparaciones, resultados negativos, resultados que desbordan los registros, etc.

C (Carry o acarreo): se pone a uno cuando se efectúa una operación que no cabe en el espacio correspondiente al resultado.

P (Paridad): se pone a uno cuando se efectúa una operación cuyo resultado contiene un número par de bits con el valor 1.

A (Auxiliar): similar al de acarreo (C), pero para las operaciones efectuadas con números en formato BCD (Binary Coded Decimal), o sea decimal codificado en binario.

Z (Cero): se pone a uno cuando se efectúa una operación cuyo resultado es cero. Ojo con esto porque a veces confunde, si se pone en cero, el resultado es distinto de cero y viceversa.

S (Signo): se pone en uno si el resultado de una operación da como resultado un valor negativo

T (Detención): si está en uno el procesador genera automáticamente una interrupción después de la ejecución de cada instrucción, lo que permite controlar paso a paso la ejecución del programa.

D (Dirección): en este caso este flag no cambia por acciones realizadas, sino que lo modificamos desde nuestro código para afectar ciertas operaciones, ya que indica la dirección a utilizar en ciertos comandos (hacia adelante o hacia atrás), como por ejemplo en comparaciones de bloques de memoria contiguos. Para modificarlo utilizamos las instrucciones std y cld, que luego veremos.

O (Overflow o desbordamiento): se pone a uno cuando se efectúa una operación cuyo resultado cambia de signo, dando un resultado incorrecto.

instrucciones básicas:

Instrucción JMP

Propósito: Salto incondicional

Sintaxis: JMP offset

Esta instrucción se utiliza para desviar el flujo de un programa sin tomar en cuenta las condiciones actuales de las banderas ni de los datos.

Ejemplo: *JMP 0401000*

Instrucción JE (JZ)

Propósito: salto condicional

Sintaxis: JE offset

Salta si es igual o salta si es cero.

El salto se realiza si la bandera Z está activada.

Ejemplo: *JE 0402000*

Instrucción JNE (JNZ)

Propósito: salto condicional

Sintaxis: JNE offset

Salta si no es igual o salta si no es cero.

El salto se efectua si la bandera Z está desactivada.

Ejemplo*: jnz 0401000*

Instrucción DEC

Propósito: Decrementar el operando

Sintaxis: DEC destino

Esta operación resta 1 al operando destino y almacena el nuevo valor en el mismo oeprando.

Ejemplo: *dec ecx*

Instrucción INC

Propósito: Incrementar el operando.

Sintaxis:INC destino

La instrucción suma 1 al operando destino y guarda el resultado en el mismo operando destino.

Ejemplo: *inc ecx*

Instrucción MOV

Propósito esta instrucción tiene dos operandos, y lo que hace es copiar el origen (representado en segundo lugar) en el de destino (en primer lugar).

Sintaxis:MOV destino,origen

Por ejemplo: *Mov eax,1*

Lo que hace ahí es mover al registro eax el numero 5

Instrucción CMP

Propósito: Comparar los operandos.

Sintaxis:CMP valor, valor

Esta instrucción compara dos valores. Generalmente se utiliza acompañada de un salto condicional de acuerdo al resultado de esa comparación.

Por ejemplo: *Cmp eax,1*

Lo que hace ahí es comparar si registro eax vale 1

Instrucción push (Push On to the Stack)

Proposito: esta instrucción resta del registro ESP la longitud de su operando que puede ser de tipo word o double word (4 u 8 bytes) y a continuación lo coloca en la pila.

Sintaxis: Push registro

Ejemplo: *Push eax*

pop (Pop a Value from the Stack)

Proposito:es la inversa de push, es decir que incrementa el registro ESP y retira el valor disponible de la pila y lo coloca donde indica el operando.

Sintaxis: Popad registro

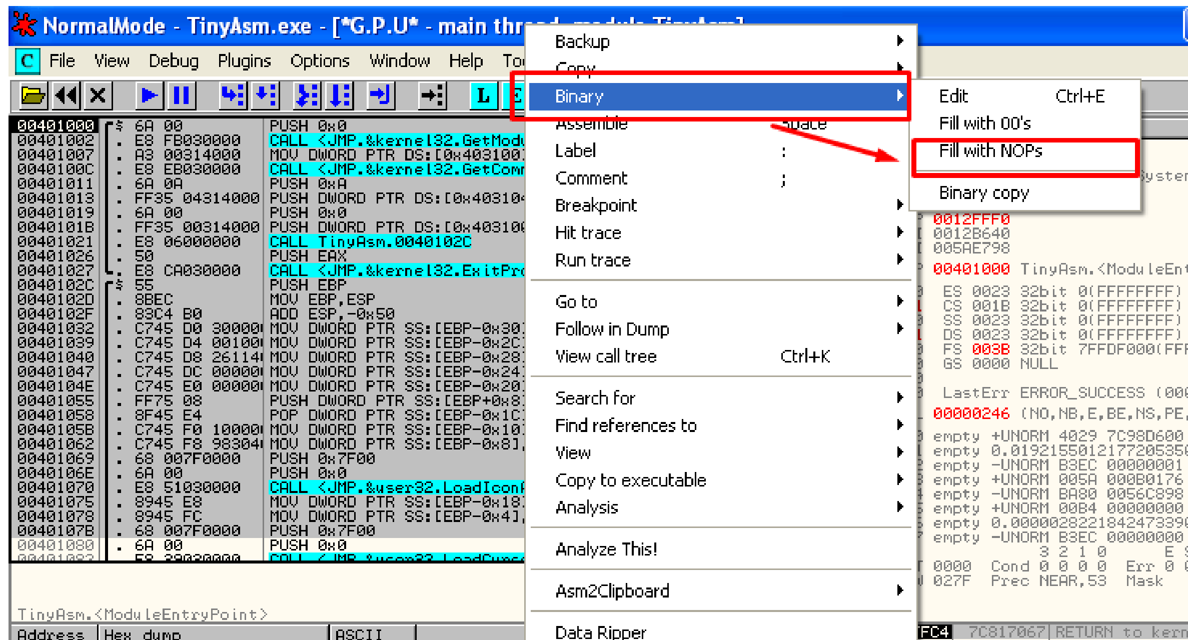
Ejemplo: Pop eax

Antes de probar con el debugger les explicare:

Al estar al inicio en la primer instrucción, es necesario sepan cómo ir ejecutando instrucción por instrucción y hay formas para ir avanzando línea por línea, la primera presionando “f7” y la segunda presionando “f8” , la primera lo que hará será pasar línea por línea hasta entrar en alguna llamada y va entrando a cada llamada que pasemos por el camino, con “f8” iremos línea por línea y aunque pasemos por alguna llamada la pasaremos por encima sin entrar en ella.

Probemos:

Ahora veamos como funcinan las instrucciones que acabamos de ver, dentro del debugger y como algunas alteran las banderas para despues tomar desiciones:



Ya teniendo el código con NOP’s, podremos empezar a probar con nuestras instrucciones y ver los cambios que hacen algunas instrucciones, comparaciones y como es que deciden saltar o no los saltos condicionales.

Escribamos el siguiente código:

Mov eax,5

Inc eac

Dec eax

Cmp eax,1

Cmp eax,5

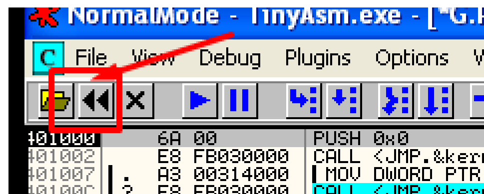
Je 401000

```
NormalMode - TinyAsm.exe - [*G.P.U* - main thread, module TinyAsm
C File View Debug Plugins Options Window Help Tools BreakPoint->
00401000   B8 05000000   MOV EAX,0x5
00401005   40            INC EAX
00401006   48            DEC EAX
00401007   83F8 01       CMP EAX,0x1
0040100A   83F8 05       CMP EAX,0x5
0040100D   90            NOP
0040100E   90            NOP
0040100F   90            NOP
00401010   90            NOP
00401011   90            NOP
00401012   90            NOP
```

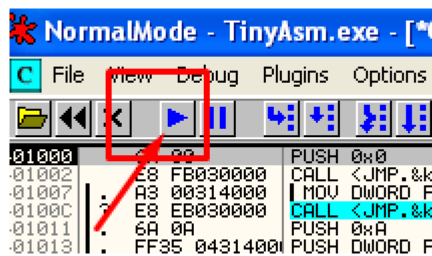
Vamos viendo como al ir paso por paso ejecutando cada instrucción, los registros van cambiando y si llegamos al salto condicional este brincara a la dirección “0401000” porque como explique el salto “JE” salta si la bandera Zero esta activa, la bandera Zero se activó al momento de hacer la comparación de eax con 5 y al ser correcto se ha activado la bandera.

Ahora reiniciemos nuestro debugger para pasar a mostrarles lo siguiente antes de empezar a crackear nuestra primer aplicación.

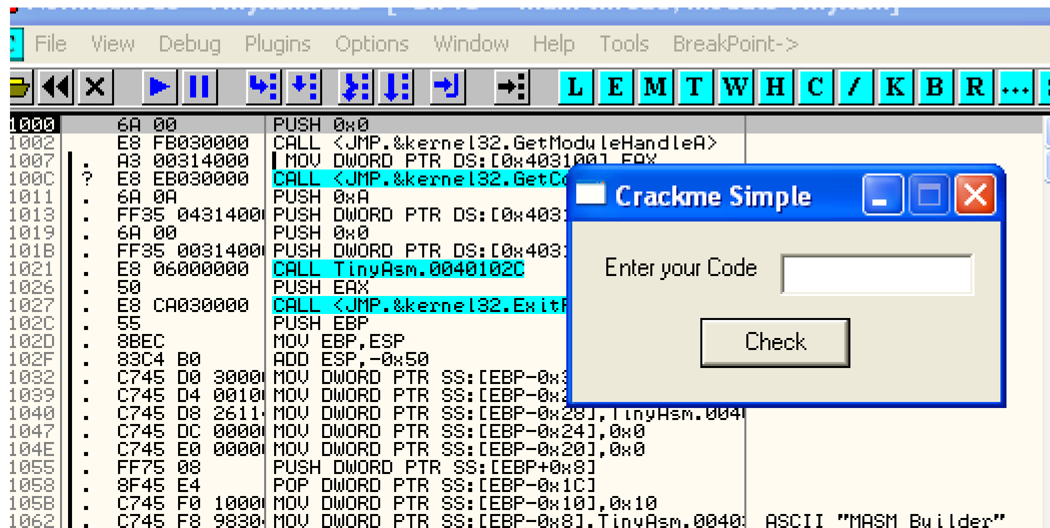
Reiniciamos con este botón:



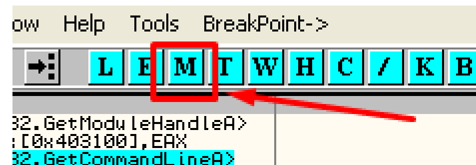
Al momento de reiniciar ahora lo que haremos será correr nuestra aplicación con el botón:



O bien presionando "f9" = run, ya teniendo corriendo nuestro ejecutable veremos como ya está cargado:



Bien aún no tocaremos nada hasta mostrarles lo siguiente, vayamos a la ventana memory map, , recuerden que arriba explique el nombre de cada botón, veamos:

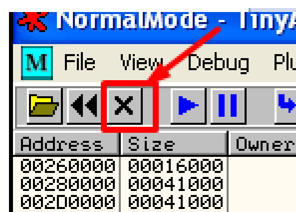


Y nos aparecerá esta ventana:

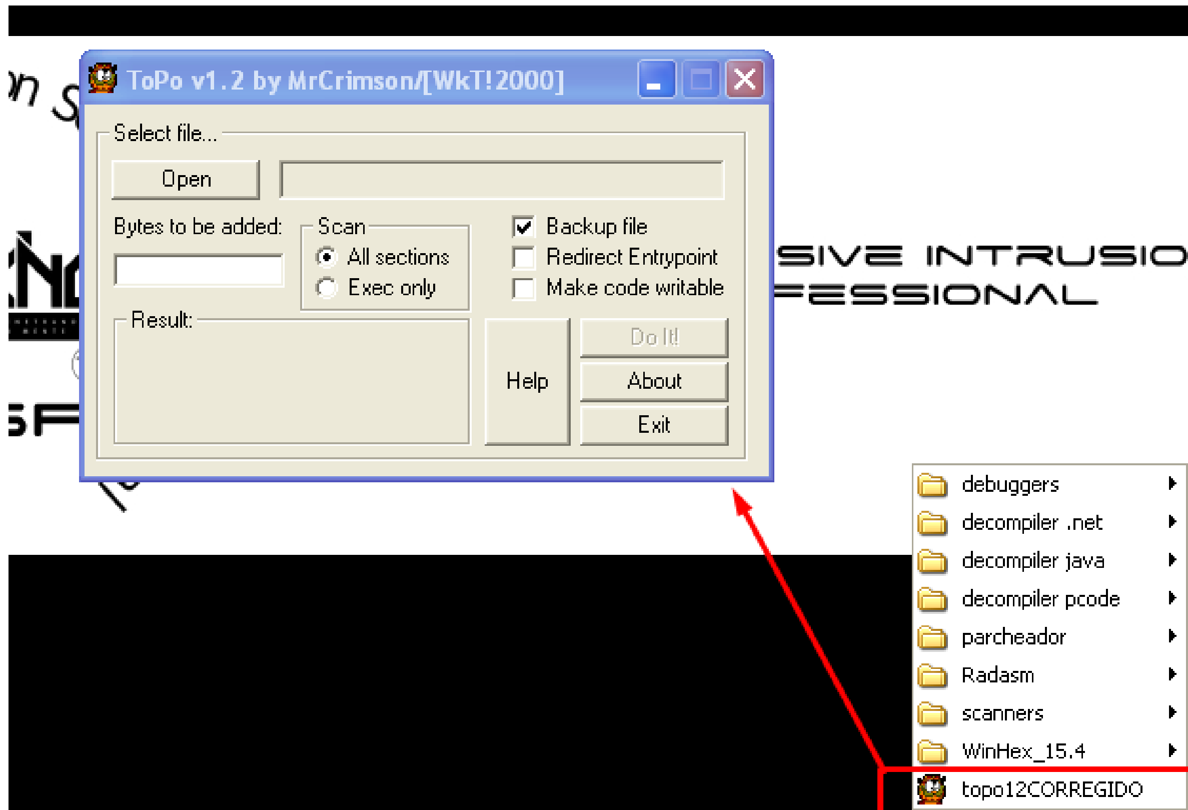
Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00260000	00016000				Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\uni
00280000	00041000				Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\loc
002D0000	00041000				Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\sor
00320000	00006000				Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\sor
00330000	00041000				Map	R	R	
00380000	00001000				Priv	RW	RW	
00390000	00001000				Priv	RW	RW	
003A0000	00005000				Priv	RW	RW	
003B0000	00003000				Map	R	R	\Device\HarddiskVolume1\WINDOWS\system32\cty
003C0000	00001000				Map	RW	RW	
003D0000	00004000				Map	RW	RW	
00400000	00001000	TinyAsm		PE header	Ima	R	RWE	
00401000	00001000	TinyAsm	.text	SFX,code	Ima	R	RWE	
00402000	00001000	TinyAsm	.rdata	data,import	Ima	R	RWE	
00403000	00001000	TinyAsm	.data		Ima	R	RWE	
00410000	00002000				Map	R E	R E	
004D0000	00002000				Map	R E	R E	
004E0000	00103000				Map	R	R	
005F0000	00053000				Map	R E	R E	
008F0000	00050000				Map	R	R	
00940000	00010000				Map	RW	RW	
00980000	00001000				Priv	RW	RW	
00A00000	00007000				Map	RW	RW	
5B150000	00001000	uxtheme		PE header	Imag	R	RWE	
5B151000	00030000	uxtheme	.text	SFX,code,im	Imag	R	RWE	
5B181000	00001000	uxtheme	.data		Imag	R	RWE	
5B182000	00004000	uxtheme	.rsrc	resources	Imag	R	RWE	
5B186000	00002000	uxtheme	.reloc		Imag	R	RWE	
746B0000	00001000	MSCTF		PE header	Imag	R	RWE	
746B1000	00042000	MSCTF	.text	SFX,code,im	Imag	R	RWE	
746F3000	00002000	MSCTF	.data	data	Imag	R	RWE	
746F5000	00004000	MSCTF	.rsrc	resources	Imag	R	RWE	
746F9000	00003000	MSCTF	.reloc		Imag	R	RWE	
77BE0000	00001000	msvcrt		PE header	Imag	R	RWE	
77BE1000	0004C000	msvcrt	.text	SFX,code,im	Imag	R	RWE	
77C2D000	00007000	msvcrt	.data	data	Imag	R	RWE	
77C34000	00001000	msvcrt	.rsrc	resources	Imag	R	RWE	
77C35000	00003000	msvcrt	.reloc		Imag	R	RWE	
77DA0000	00001000	advapi32		PE header	Imag	R	RWE	

Lo que está en el rectángulo, son las secciones del ejecutable, tenemos la primer sección que es la cabecera del programa la cual contiene la estructura PE, después tenemos la sección .text o sección .code que es la sección donde está el código del programa, donde hace un momento estábamos ejecutando código, si ven en el lado izquierdo están las dirección y el tamaño de cada sección, un poco mas abajo esta la sección .rdata que es donde se encuentra la IAT.

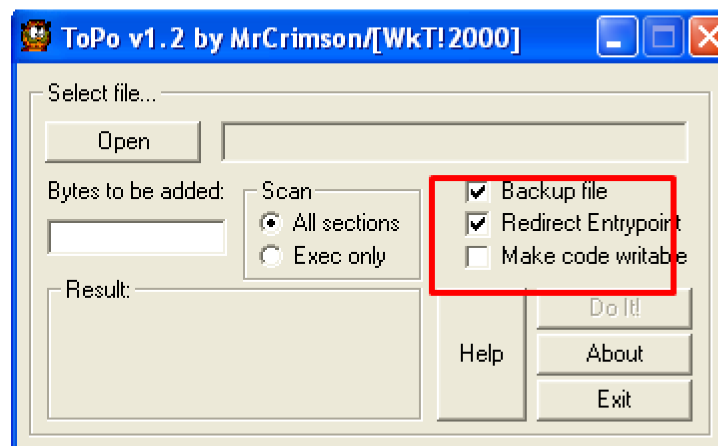
Cerramos el proceso para pasar a lo siguiente:



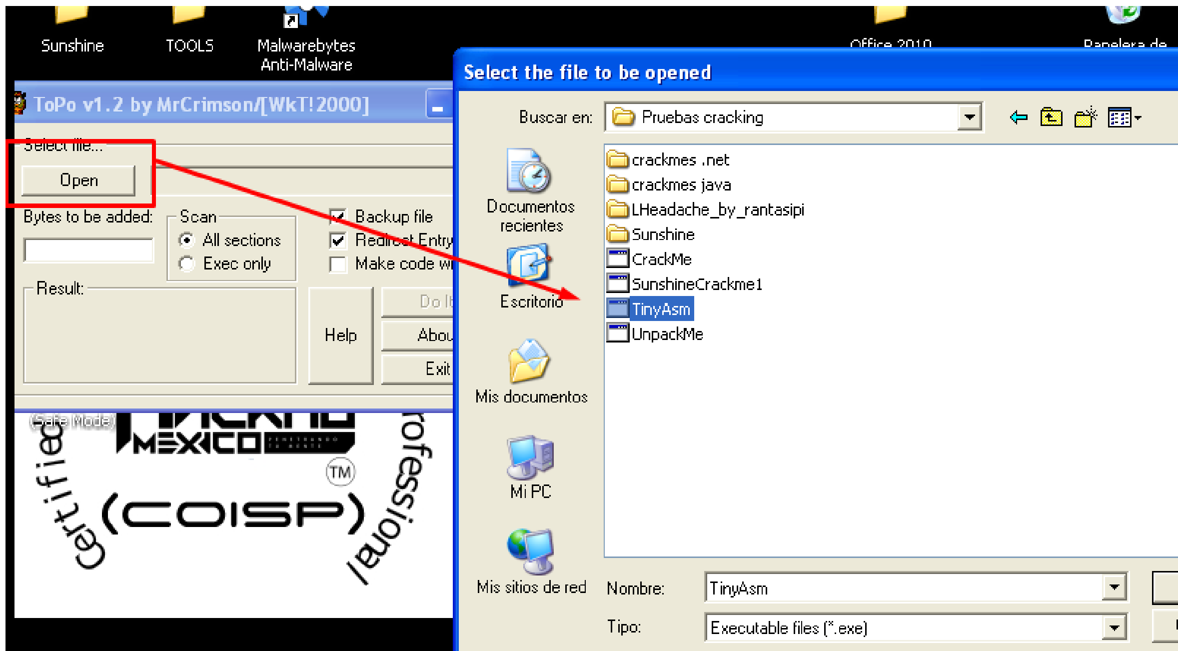
Ahora bien vamos a pasar a la parte donde creamos una sección de código vacía, para ello abrimos topo, que es una de nuestras herramientas dentro de la máquina virtual, veamos:



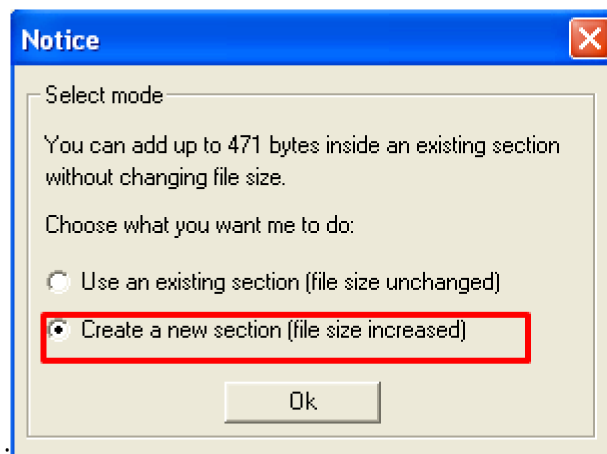
Una vez abierto, veremos algunas opciones, las cuales para agregar una sección deberemos hacer lo siguiente:



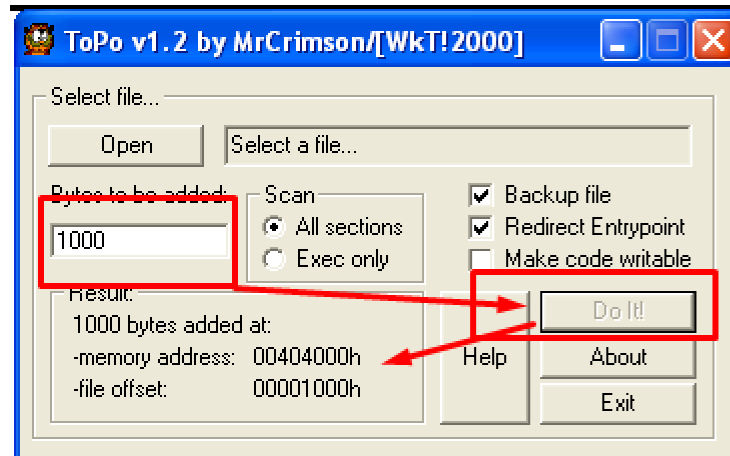
Que la primera es para que cuando agreguemos una sección al binario, guarde un backup del mismo en caso de habernos equivocado, la opción de abajo “redirect entrypoint” es para que nuestro ejecutable ejecute primero nuestra sección creada y ya será responsabilidad de nosotros regresar a la dirección donde iniciaba normalmente el ejecutable, seguimos y seleccionamos “open” y presionamos ok en el siguiente mensaje y buscamos nuestro binario:



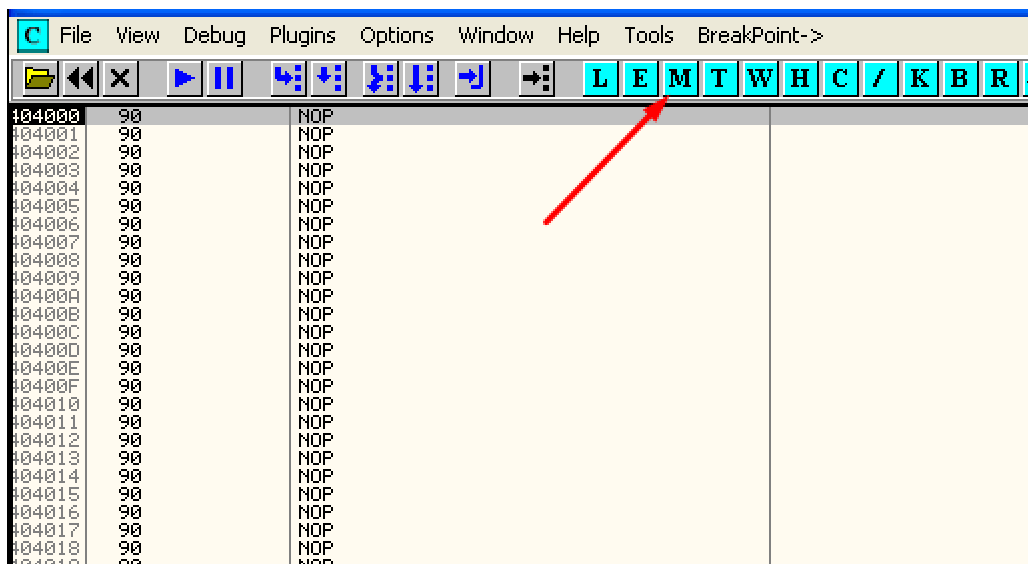
Despues de seleccionar el archivo con doble click, nos saldrá esta ventana, la cual nos preguntara, si quieremos agregar una nueva sección o utilizar una existente y me dice que tengo 471 bytes disponibles en caso de querer usar una sección ya existente pues lo que hizo fue buscar huecos vacios, pero lo que quiero es agregar una sección nueva, asi que seleccionamos esa opción y le damos ok:



Ahora debemos seleccionar cuantos bytes queremos agregar, ahmm digamos que quiero agregar 1000 bytes, metamos esos datos y demos en el botón, “do IT”:



Abrimos de nuevo nuestro ejecutable con olly debugger como ya les explique anteriormente y una vez cargado, veremos que el inicio ya no es como antes ahora es una zona vacia pues como les dije al tildar la opción “Redirect Entrypoint” este nos ha puesto al principio nuestra zona creada vamos a la sección Memory, con el botón “M” y veremos nuestra nueva sección de código creada:



Si lo notan ya está la sección creada de tamaño “1000” bytes, ahora bien que podemos hacer con eso? Muchísimo, pues lo que podríamos hacer es fácilmente hacer un injerto de código, lo mostrado es prácticamente lo que haría in virus cuando infecta a un programa solo que esta vez lo hicimos a mano con la herramienta, esta sección nueva nos puede servir de muchísimo al momento de querer crackear un programa y es esencial tenerlo en cuenta, antes de comenzar a atacar el programa vamos a redirigir el funcionamiento a la entrada que tenia el programa normalmente, pera ello recordemos vamos a usar un salto, uno incondicional, que salte siempre, recodemos que el salto incondicional es “JMP” y su sintaxis es “jmp offset”por lo tanto escribimos:

Ya teniendo la instrucción ahora pasamos a presionar “f8” para ejecutar esa instrucción y que nos llevara a:

Bien ya hemos dominado un poco el funcionamiento del debugger, los saltos condicionales e incondicionales, escribir instrucciones en asm dentro del debugger, crear secciones de codigo, ahora vamos a presionar “f9” para que corra nuestra aplicación dentro del debugger:

Bien ahora les explico, al momento que ejecutamos un programa dentro de nuestro debugger, nosotros ya tenemos completo control sobre él, lo que hace este es ejecutar todas las instrucciones de arriba hacia abajo pasando por cada una de las instrucciones y guiándose de los saltos condicionales para tomar el flujo del programa.

Cuando hablamos de software con limitaciones o protecciones, es lógico que en alguna parte del programa hace comprobaciones de la contraseña, serial, licencia y demás cosas dependiendo la

protección, lo importante es encontrar aquellas zonas encargadas de hacer las comprobaciones e ir paso a paso instrucción por instrucción tratando de averiguar donde hace los chequeos es un tema muy largo; por lo cual se han desarrollado algunas técnicas para encontrar fácilmente la rutina o la zona cercana donde se hacen las comprobaciones o chequeos.

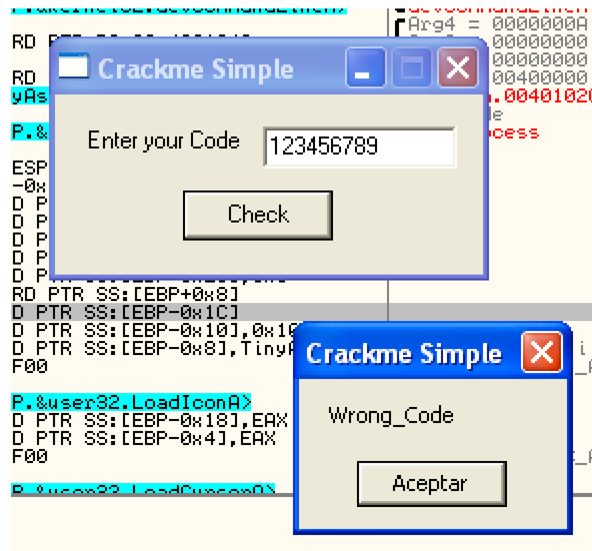
Las dividí en tres tipos:

Cazando las api's que utiliza (es necesario conocer una gran parte de ellas)

Metodo String Reference's (buscando las cadenas de texto del programa)

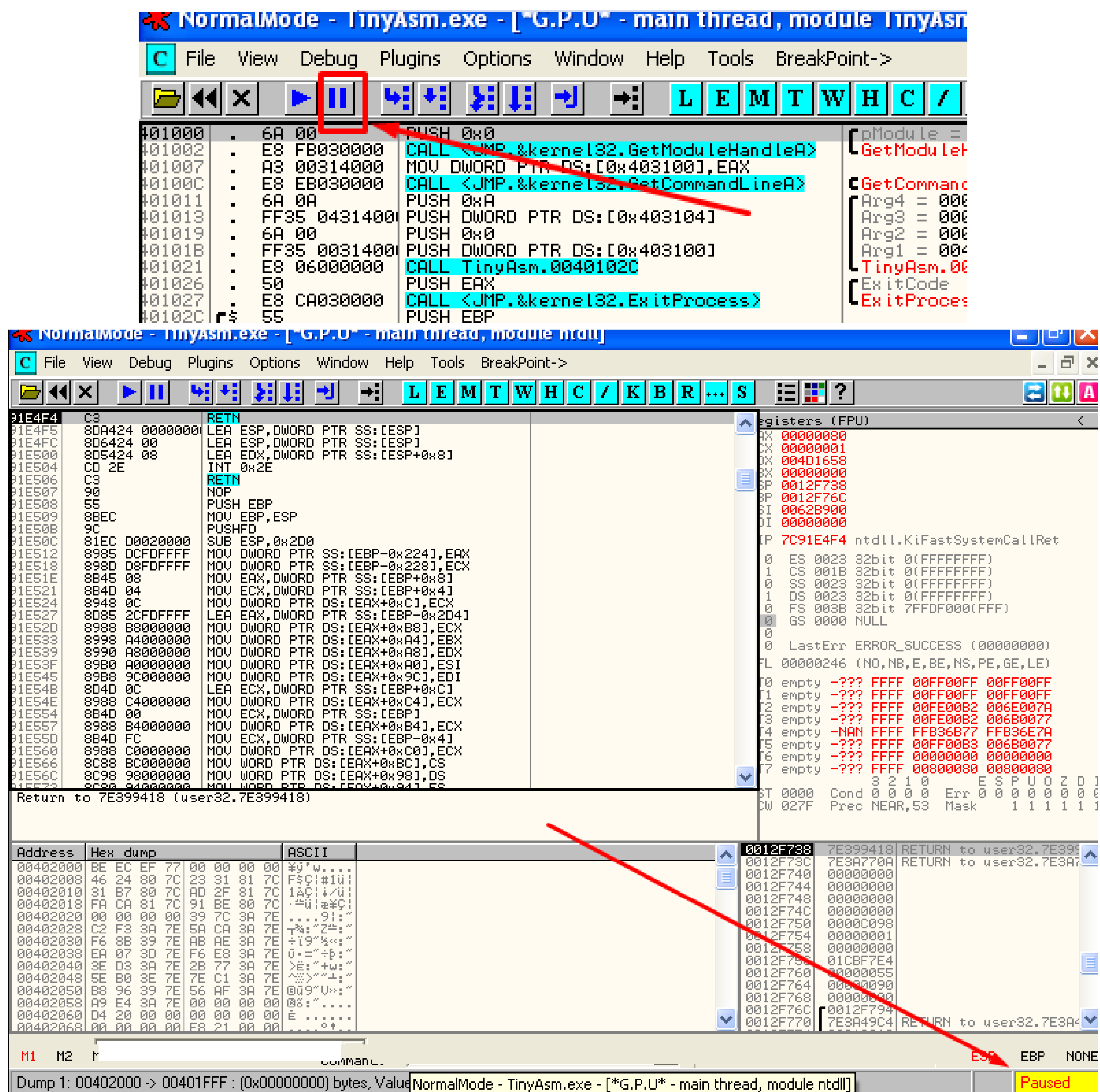
Cazando los mensajes (funciona cuando la limitación o verificación nos muestra algún mensaje), esta es la más común ya que la mayoría del software co limitaciones siempre nos muestra algún mensaje del tipo MsgBox.

Vamos a comenzar cazando los mensajes, vemos la ventana que nos pedia alguna clave e ingresamos cualquier clave y presionamos "check" haber que nos dice, veamos:



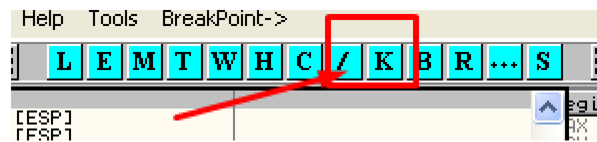
Ya que el mensaje nos ha salido, les explico, cuando un mensaje es mostrado es porque Windows Ya llamo a la API (MessageBoxA) y este ha mostrado el mensaje el cual al seguir el mensaje en pantalla aún quiere decir que la ejecución del programa sigue estando ejecutando la API, no ha salido de la API, pues para que este salga de la API y siga su ejecución normal es necesario presionar el botón aceptar.

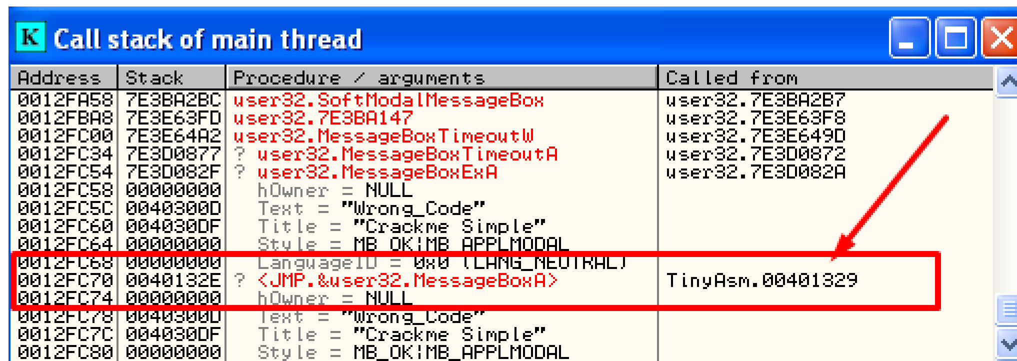
Pero antes de eso, volvemos al debugger y presionamos “pause”:



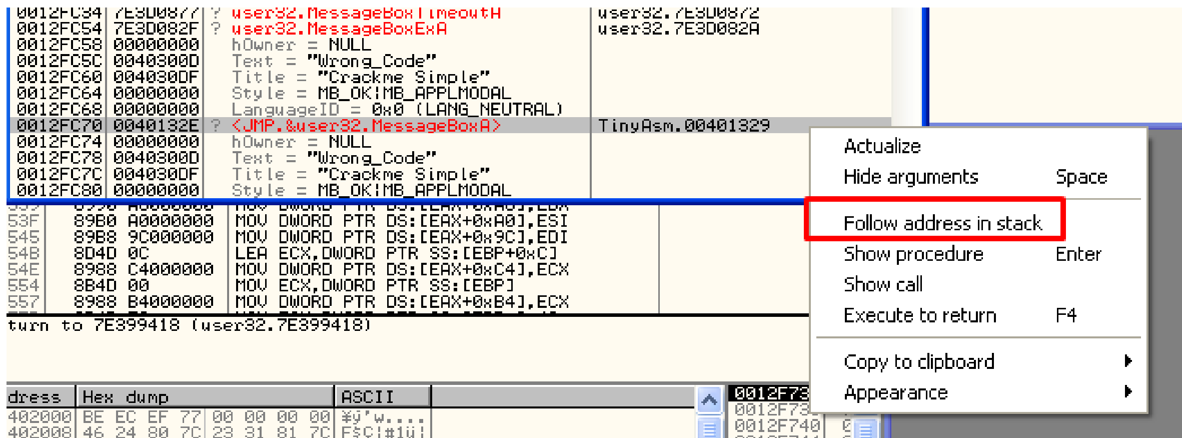
Ahora está detenida la ejecución del debugger lo cual si recuerdan les comente que la ejecución seguía dentro de la API que nos mostraba el mensaje malo, entonces para poder ver las ultimas llamadas donde entro nuestro debugger presionamos una de los botones, en específico el botón "K" que si recuerdan:

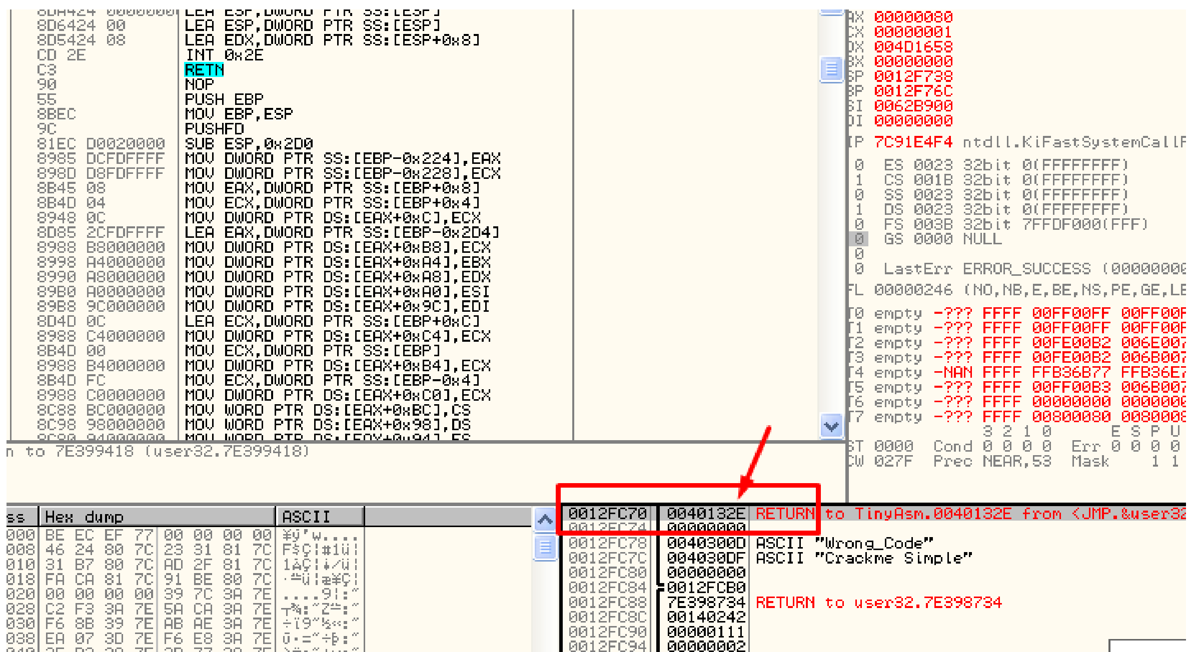
"K: call stack of main thread, muestra los distintos calls (llamadas) a los que vamos entrando"



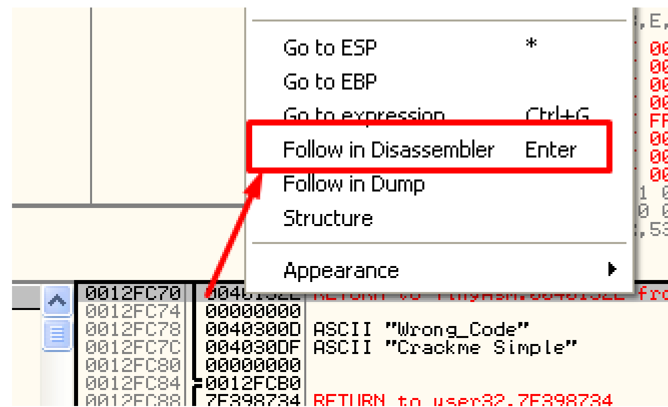


Veremos en la ventana en las ultimas direcciones, la que esta en color rojo es una llamada a una API, en este caso es MessageBoxA y de lado derecho en la parte de "called from" nos está diciendo que es llamado del offset "0401329", la forma sencilla de llegar ahí es seleccionamos (la dirección que está dentro del rectángulo haya arriba), damos click derecho y "follow address in stack", lo cual nos manda la dirección al stack o pila, recuerdan cual era el stack no?





La primer línea del stack está la dirección de donde fue llamada esa API lo cual es la del mensaje malo, ahora seleccionamos esa primer dirección (la que está dentro del rectángulo y señalada) y damos click derecho y seleccionamos "follow in disassembler" , veamos:



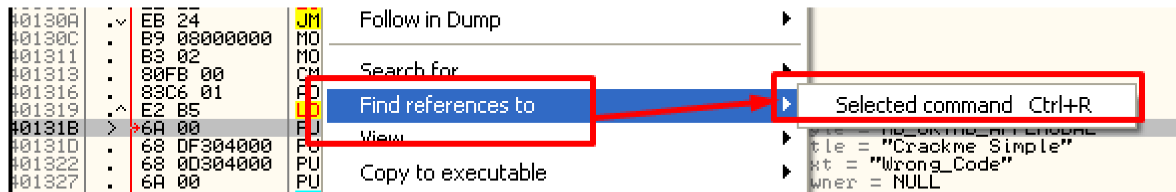
Lo cual nos mandara a la parte del disassembler a la dirección, mostrándonos toda la rutina que es la encargada de mostrar el mensaje malo:

```
0130A  .v EB 24        JMP SHORT TinyAsm.00401330
0130C  .  B9 08000000  MOV ECX,0x8
01311  .  B3 02        MOV BL,0x2
01313  .  80FB 00      CMP BL,0x0
01316  .  83C6 01      ADD ESI,0x1
01319  .^ E2 B5        LOOPD SHORT TinyAsm.004012D0
0131B  >  6A 00        PUSH 0x0                            ┌Style = MB_OK|MB_APPLMODAL
0131D  .  68 DF304000  PUSH TinyAsm.004030DF               │Title = "Crackme Simple"
01322  .  68 0D304000  PUSH TinyAsm.0040300D               │Text = "Wrong_Code"
01327  .  6A 00        PUSH 0x0                            │hOwner = NULL
01329  .  E8 9E000000  CALL <JMP.&user32.MessageBoxA>      └MessageBoxA
0132E  .v EB 13        JMP SHORT TinyAsm.00401343
01330  >  6A 00        PUSH 0x0                            ┌Style = MB_OK|MB_APPLMODAL
01332  .  68 DF304000  PUSH TinyAsm.004030DF               │Title = "Crackme Simple"
01337  .  68 18304000  PUSH TinyAsm.00403018               │Text = "Yes_Succes"
0133C  .  6A 00        PUSH 0x0                            │hOwner = NULL
0133E  .  E8 89000000  CALL <JMP.&user32.MessageBoxA>      └MessageBoxA
01343  >  6A 00        PUSH 0x0                            ┌ExitCode = 0x0
01345  .  E8 AC000000  CALL <JMP.&kernel32.ExitProcess>    └ExitProcess
0134A  >  90           NOP
0134B  .  90           NOP
0134C  .  90           NOP
```

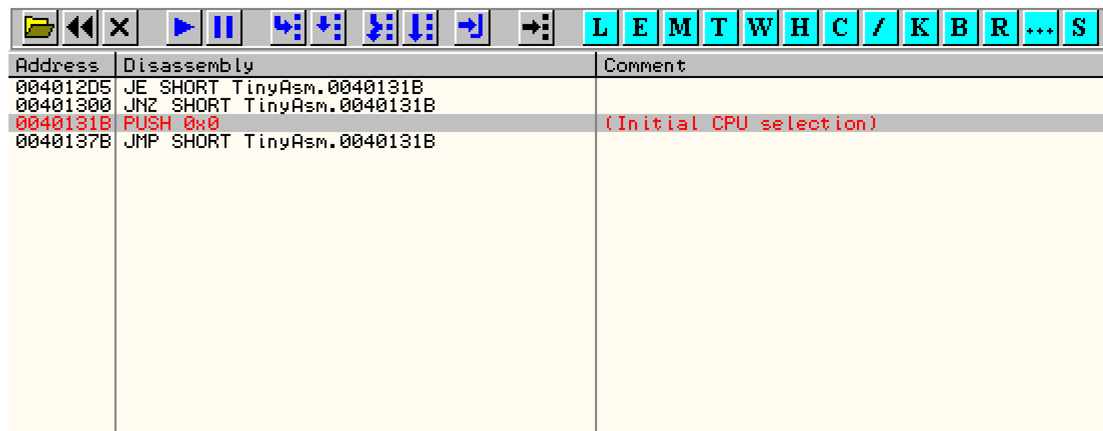
Si vemos, caimos dejabo de donde nos muestra el mensaje (call MessageBox) y si analizamos , tenemos el codigo para mostrar dos mensajes, uno nos indica “wrong_code” y el otro “Yes_succes” , para poder saber de donde inicia la rutina, nuestro debugger nos facilita con el signo “>”, estándo ubicado al principio del MsgBox que nos muestra el mensaje malo.

```
313  .  80FB 00      CMP BL,0x0
316  .  83C6 01      ADD ESI,0x1
319  .^ E2 B5        LOOPD SHORT TinyAsm.004012D0
31B  >  6A 00        PUSH 0x0                           ┌Style = MB_OK|MB_APPLMODAL
31D  .  68 DF304000  PUSH TinyAsm.004030DF              │Title = "Crackme Simple"
322  .  68 0D304000  PUSH TinyAsm.0040300D              │Text = "Wrong_Code"
327  .  6A 00        PUSH 0x0                           │hOwner = NULL
329  .  E8 9E000000  CALL <JMP.&user32.MessageBoxA>     └MessageBoxA
32E  .v EB 13        JMP SHORT TinyAsm.00401343
330  >  6A 00        PUSH 0x0                           ┌Style = MB_OK|MB_APPLMODAL
```

Lo selecionamos y presionamos click derecho y “Find references to” y seleccionamos “selected command”

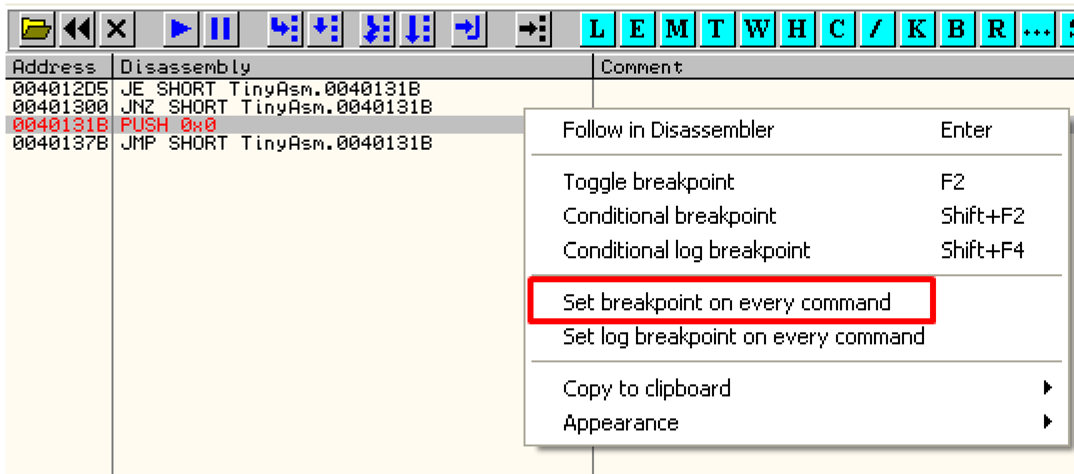


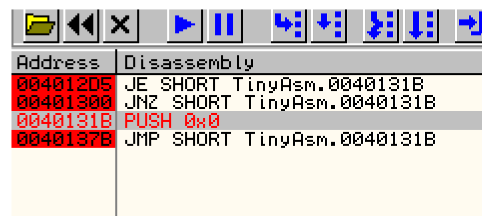
Lo cual nos mostrara todas las referencias que llaman a esa dirección :



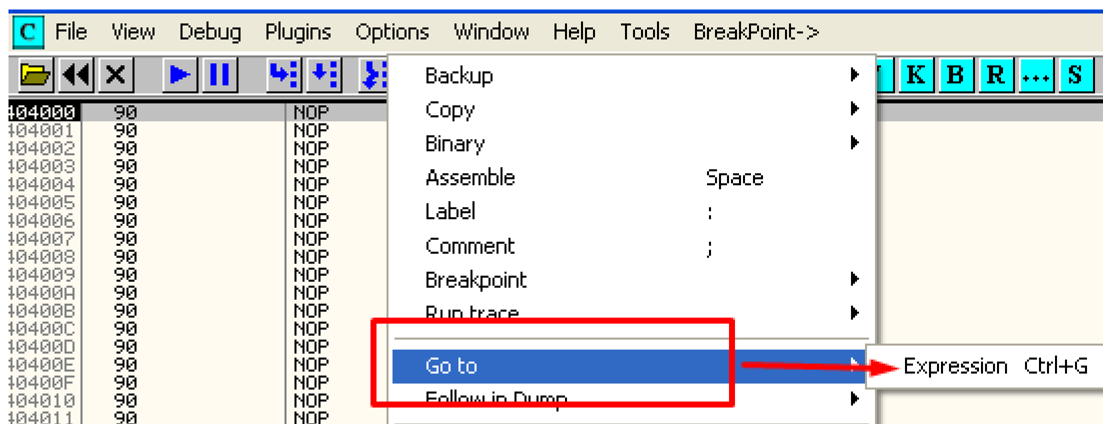
Como ya sabemos esos son saltos condicionales, en diferentes direccione, lo cual quiere decir que de esas tres direcciones hace comprobaciones y decide si mandarnos o no algún mensaje malo, lo que haremos ahora será poner un punto de ruptura (BreakPoint) para que cuando el programa pase por alguno de esos saltos que nos mandan al mensaje malo, este pare su ejecución y ver si podríamos esquivar esto logrando que salga nuestro mensaje correcto.

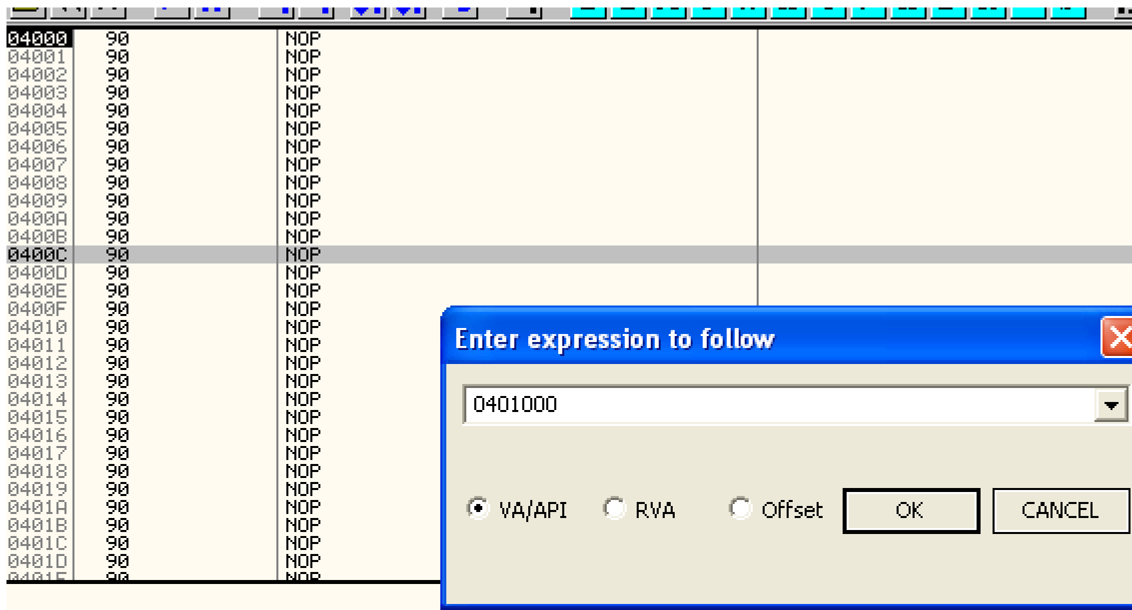
Hacemos click derecho y seleccionamos “Set BreakPoint on every command” que nos pondrá un breakpoint en todas las direcciones que encontró.





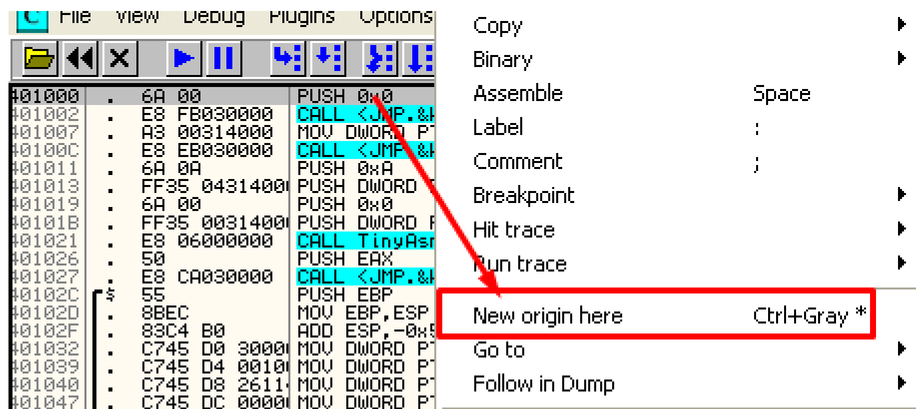
Cuando el breakpoint es colocado las direcciones se pintan en rojo, indicándonos que el Breakpoint está colocado, ahora bien reiniciemos el debugger y si recordamos volvimos a caer en la zona creada anteriormente, y no se guardó nuestro salto “jmp” porque solo fue un cambio en memoria, mas adelante les mostrare como guardar los cambios pero bueno ahora vamos a ir al principio del codigo antes de que modificaramos, fácilmente seleccionando click derecho “goto” y seleccionamos “expression”:





Nos parece la casilla e ingresamos, que si recordamos, la dirección era “0401000”.

Veremos estamos donde anteriormente, en la línea de codigo damos click derecho de nuevo y seleccionamos “new origin here”:



Ya estando en la primer línea de código, podemos pasar a la ventana breakpoints para ver si siguen activos que podemos acceder presionando el botón “B”:

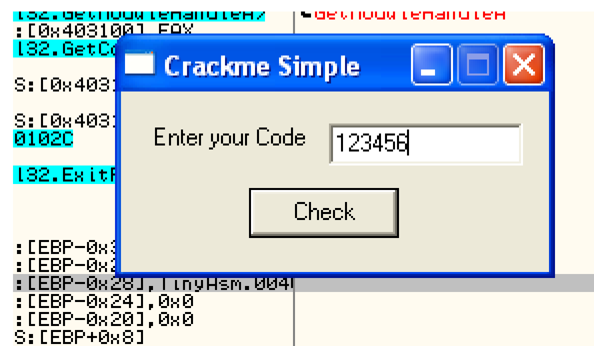
```
C File View Debug Plugins Options Window Help Tools BreakPoint->
L E M T W H C / K B R ... S

401000    .  6A 00          PUSH 0x0                                   pModule = NULL
401002    .  E8 FB030000    CALL <JMP.&kernel32.GetModuleHandleA>      GetModuleHandleA
401007    .  A3 00314000    MOV DWORD PTR DS:[0x403100],EAX
40100C    .  E8 EB030000    CALL <JMP.&kernel32.GetCommandLineA>       GetCommandLineA
401011    .  6A 0A          PUSH 0xA                                   Arg4 = 0000000A
401013    .  FF35 0431400   PUSH DWORD PTR DS:[0x403104]               Arg3 = 00000000
401019    .  6A 00          PUSH 0x0                                   Arg2 = 00000000
40101B    .  FF35 0031400   PUSH DWORD PTR DS:[0x403100]               Arg1 = 00000000
401021    .  E8 06000000    CALL TinyAsm.0040102C                      TinyAsm.0040102C
401026    .  50             PUSH EAX                                   ExitCode
401027    .  E8 CA030000    CALL <JMP.&kernel32.ExitProcess>           ExitProcess
40102C    $  55             PUSH EBP
40102D    .  8BEC           MOV EBP,ESP
40102F    .  83C4 B0        ADD ESP,-0x50
401032    .  C745 D0 3000   MOV DWORD PTR SS:[EBP-0x30],0x30
401039    .  C745 D4 0010   MOV DWORD PTR SS:[EBP-0x2C],0x1000
401040    .  C745 D8 2611   MOV DWORD PTR SS:[EBP-0x28],TinyAsm.004
401047    .  C745 DC 0000   MOV DWORD PTR SS:[EBP-0x24],0x0
40104E    .  C745 E0 0000   MOV DWORD PTR SS:[EBP-0x20],0x0
401055    .  FF75 08        PUSH DWORD PTR SS:[EBP+0x8]
401058    .  8F45 E4        POP DWORD PTR SS:[EBP-0x1C]
40105B    .  C745 F0 1000   MOV DWORD PTR SS:[EBP-0x10],0x10
401062    .  C745 F8 9830   MOV DWORD PTR SS:[EBP-0x8],TinyAsm.0040  ASCII "MASM Builder"
401069    .  68 007F0000    PUSH 0x7F00                                RsrcName = IDI_APPLICATION
40106E    .  6A 00          PUSH 0x0                                   hInst = NULL
401070    .  E8 51030000    CALL <JMP.&user32.LoadIconA>               LoadIconA
401075    .  8945 E8        MOV DWORD PTR SS:[EBP-0x18],EAX
```

Breakpoints

Address	Module	Active	Disassembly
004012D5	TinyAsm	Always	JE SHORT TinyAsm.0040131B
00401300	TinyAsm	Always	JNZ SHORT TinyAsm.0040131B
0040137B	TinyAsm	Always	JMP SHORT TinyAsm.0040131B

Hay podemos ver la lista de breakpoint que habíamos colocado, ahora si estamos listos, presionamos “f9” para darle ejecutar a la aplicación :



Metemos cualquier número y damos “check”, a ver en cual de nuestros puntos de ruptura para la ejecución:

```
40136A  .  33D8           XOR EBX,EAX
40136C  .  47             INC EDI
40136D  .^ E2 F5          LOOPD SHORT TinyAsm.00401364
40136F  .  2B05 0930400   SUB EAX,DWORD PTR DS:[0x403009]
401375  .^ 0F84 3AFFFFF   JE TinyAsm.004012B5
40137B  .^ EB 9E          JMP SHORT TinyAsm.0040131B
40137D  .  6A 00          PUSH 0x0                              ExitCode = 0x0
40137F  .  E8 72000000    CALL <JMP.&kernel32.ExitProcess>      ExitProcess
401384     90             NOP
401385  >  EB 15          JMP SHORT TinyAsm.0040139C
```

Vemos que paro la ejecución en un salto “JMP” lo cual es un salto incondicional y nos lleva al mensaje malo, pero arriba de ese vemos un salto “JE” salta si el flag zero está activado, de cualquier forma ese salto es el que no evita el mensaje malo pongamos un breakpoint en esa instrucción seleccionamos el salto y presionamos “f2”, reiniciemos y hagamos lo mismo, hasta que pare en ese salto condicional “JE”

al hacer el mismo procedimiendo ya con el BreakPoint colocado, damos “ejecutar” , escribimos la clave y damos en “Check” ahora vemos que ha parado en el salto “je”:

```
40132E  .v EB 13          JMP SHORT TinyAsm.00401343
401330  >  6A 00          PUSH 0x0
401332     68 DF304000    PUSH TinyAsm.004030DF
401337     68 18304000    PUSH TinyAsm.00403018
40133C     6A 00          PUSH 0x0
40133E     E8 89000000    CALL <JMP.&user32.MessageBoxA>
401343     6A 00          PUSH 0x0
401345     E8 AC000000    CALL <JMP.&kernel32.ExitProcess>
40134A     90             NOP
40134B     90             NOP
40134C     90             NOP
40134D     90             NOP
40134E     90             NOP
40134F     BF 7E124000    MOV EDI,TinyAsm.0040127E
401354     BB F2FCAA5D    MOV EBX,0x5DAAFCF2
401359     B9 4A134000    MOV ECX,TinyAsm.0040134A
40135E     81E9 7E12400   SUB ECX,TinyAsm.0040127E
401364     8B07           MOV EAX,DWORD PTR DS:[EDI]
401366     33C3           XOR EAX,EBX
401368     D1E3           SHL EBX,1
40136A     33D8           XOR EBX,EAX
40136C     47             INC EDI
40136D   ^ E2 F5          LOOPD SHORT TinyAsm.00401364
40136F  .  2B05 0930400   SUB EAX,DWORD PTR DS:[0x403009]
401375  .^ 0F84 3AFFFFF   JE TinyAsm.004012B5
40137B  .^ EB 9E          JMP SHORT TinyAsm.0040131B
40137D  .  6A 00          PUSH 0x0
```

Podemos ver hemos parado en el salto condicional, aquí podríamos cambiarlo para que en vez de mostrar el mensaje malo siga la ejecución por otro lado que es lógico será seguir la ejecución por donde es correcto, podríamos convertir ese salto en un salto indirecto “JMP”

```
0136F  .   2B05 0930400  SUB EAX,DWORD PTR DS:[
01375  ^   E9 3BFFFFFF   JMP TinyAsm.004012B5
0137A      90            NOP
0137B  .^  EB 9E         JMP SHORT TinyAsm.0040
```

Podemos ver como han cambiado algunos opcode del lado izquiero pero bueno seguimos la ejecución presionando “f9” porque si recuerdan hay breakpoints ya colocados y veamos haber si para en algún otro lugar, damos run y veamos:

```
4012D5  v  74 44          JE SHORT TinyAsm.0040131B
4012D7  .  83C6 01        ADD ESI,0x1
4012DA  ^  E2 F4          LOOPD SHORT TinyAsm.004012D0
4012DC  .  90             NOP
4012DD  .  BF 2C304000    MOV EDI,TinyAsm.0040302C        ASCII "12233"
4012E2  .  BE 23304000    MOV ESI,TinyAsm.00403023        ASCII "zFE[E^M."
4012E7  .  B9 07000000    MOV ECX,0x7
4012EC  >  8A07           MOV AL,BYTE PTR DS:[EDI]
4012EE  .  8A1E           MOV BL,BYTE PTR DS:[ESI]
4012F0  .  FEC3           INC BL
4012F2  .  321D 34304000  XOR BL,BYTE PTR DS:[0x403034]
4012F8  .  3205 35304000  XOR AL,BYTE PTR DS:[0x403035]
4012FE  .  2AC3           SUB AL,BL
401300  v  75 19          JNZ SHORT TinyAsm.0040131B
401302  .  83C6 01        ADD ESI,0x1
401305  .  83C7 01        ADD EDI,0x1
401308  .^ E2 E2          LOOPD SHORT TinyAsm.004012EC
40130A  .v EB 24          JMP SHORT TinyAsm.00401330
40130C  .  B9 08000000    MOV ECX,0x8
401311  .  B3 02          MOV BL,0x2
401313  .  80FB 00        CMP BL,0x0
401316  .  83C6 01        ADD ESI,0x1
401319  .^ E2 B5          LOOPD SHORT TinyAsm.004012D0
40131B  >  6A 00          PUSH 0x0                        Style = MB_OK|MB_APPLMODA
40131D  .  68 DF304000    PUSH TinyAsm.004030DF           Title = "Crackme Simple"
401322  .  68 0D304000    PUSH TinyAsm.0040300D           Text = "Wrong_Code"
401327  .  6A 00          PUSH 0x0                        hOwner = NULL
```

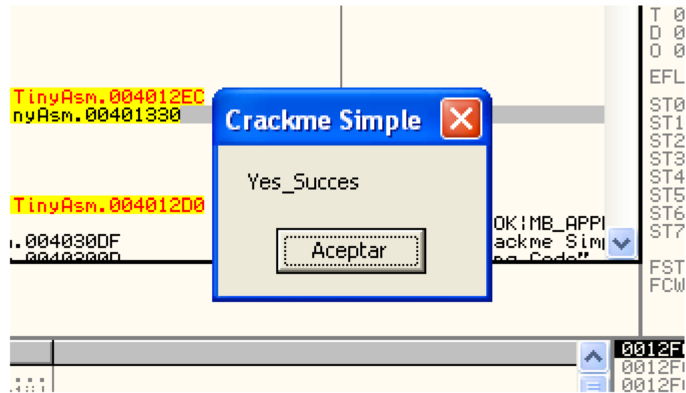
Nos ha parado la ejecución en otro salto condicional que como vemos nos indica que no será tomado pero debajo vemos un loopd, aunque el salto en ese momento no sea tomado vemos que en algún momento podría ser tomado y mandarnos al mensaje de error así que lo borraremos con un NOP, que es una instrucción no hace nada y se usa para llenar huecos, escribimos encima hasta quedar así y quitamos el BP presionando “f2” encima del BP:

```
4012CB  .  B9 07000000  MOV ECX,0x7
4012D0  >  8A1E         MOV BL,BYTE PTR DS:[ESI]
4012D2  .  80EB 00      SUB BL,0x0
4012D5     90           NOP
4012D6     90           NOP
4012D7  .  83C6 01      ADD ESI,0x1
4012DA  .^ E2 F4        LOOPD SHORT TinyAsm.004012D0
4012DC  .  90           NOP
4012DD  .  BF 2C304000  MOV EDI,TinyAsm.0040302C
4012E2  .  BE 23304000  MOV ESI,TinyAsm.00403023
4012E7  .  B9 07000000  MOV ECX,0x7
4012EC  >  8A07         MOV AL,BYTE PTR DS:[EDI]
4012EE  .  8A1E         MOV BL,BYTE PTR DS:[ESI]
```

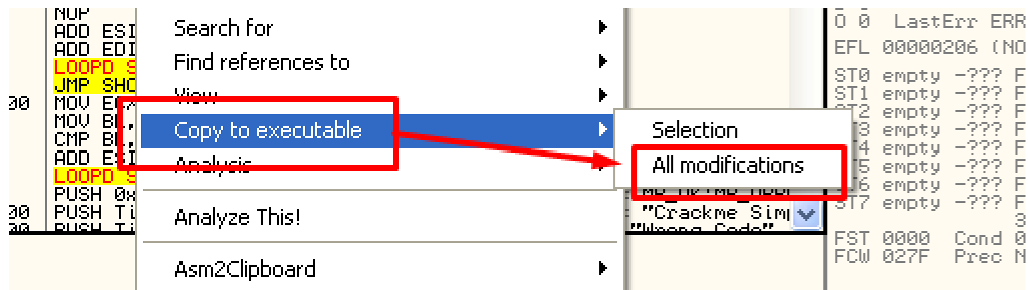
Seguimos corriendo nuestro programa con “f9” haber si nos para en el siguiente salto condicional:

Nos para en otro salto condicional esta vez es un “JNZ” como ya sabemos podríamos pasarlo cambiando la bandera, pero queremos empezar a hacer las modificaciones para empezar a guardar cambios, asi que si queremos que nunca tome ese salto, podríamos borrarlo o cambiarlo, podríamos utilizar de nuevo NOP, borrar esa instrucción:

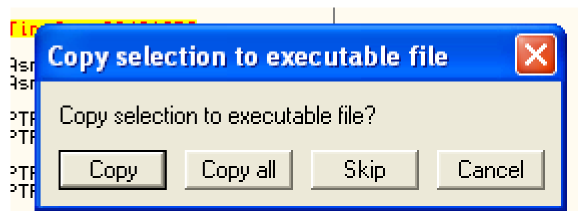
Debajo veremos un loopd y abajo un “jmp” donde he colocado un breakpoint (marcado en rojo para referencia) que si ponemos atención veriamos que este nos manda al mensaje del chico bueno, probemos demos run hasta que nos muestre:

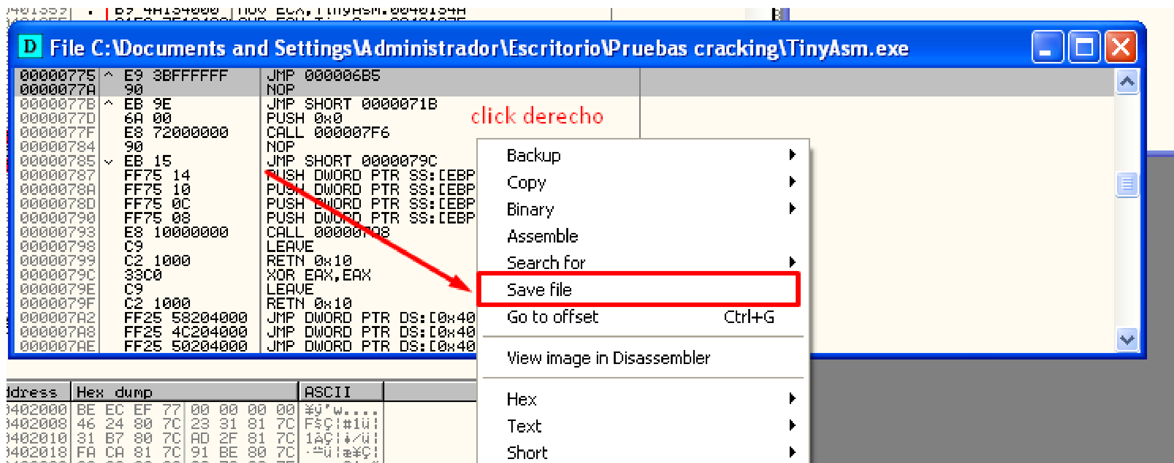


Veamos que es muy simple lograr cambiar los saltos condicionales, lo cual no se ve tan fino y existen mejores formas, como lograr entender el comportamiento, pero el objetivo se ha logrado que es lograr burlar la protección, uno de los pasos importantes siguientes es crear nuestro parche y tratare de mostrar algunas formas de hacerlo, empezaremos desde nuestro debugger, para guardar los cambios efectuados (que recordemos solo son en memoria) , bien regresemos al desassembler y hacemos click derecho y seleccionamos "copy to executable" y después "all modifications" para que asi nos guarde todas las modificaciones.

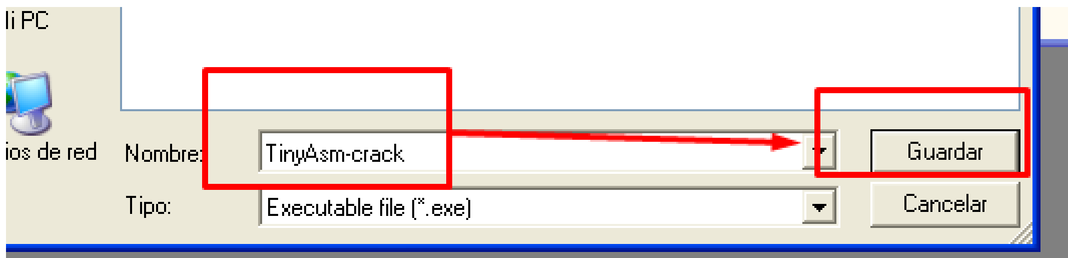


Despues en el mensaje presionamos "copy all":

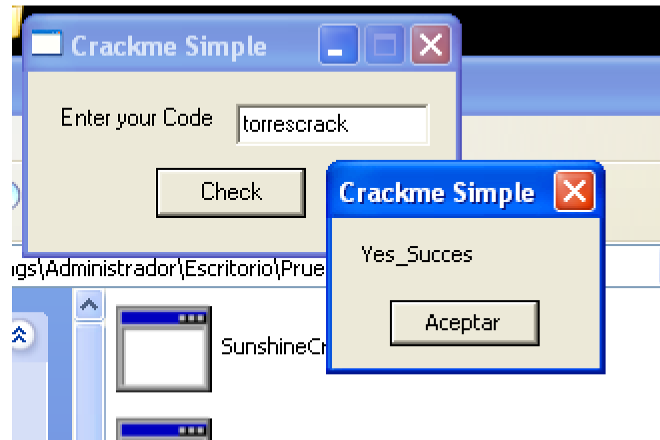




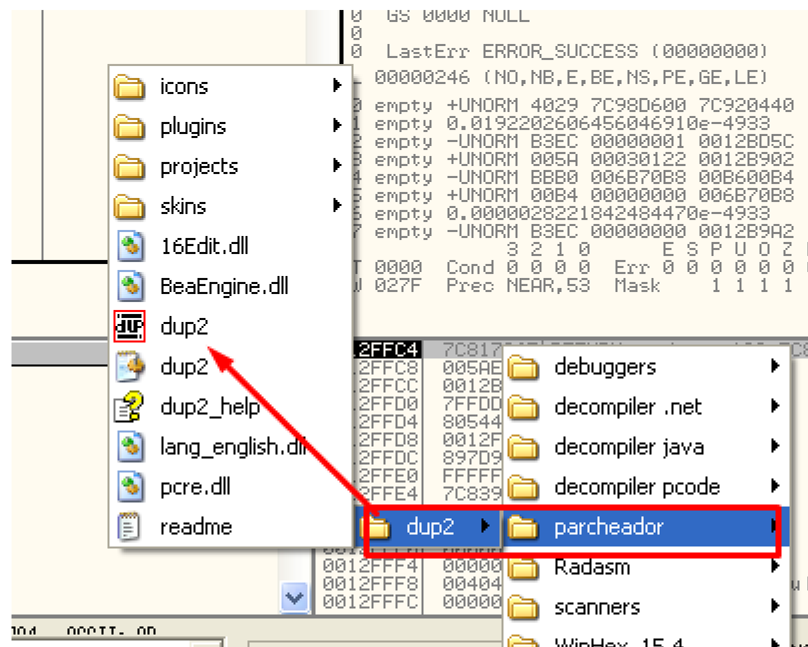
En la ventana que nos aparece damos click derecho y en el menú seleccionamos "save file" y lo guardamos con otro nombre:



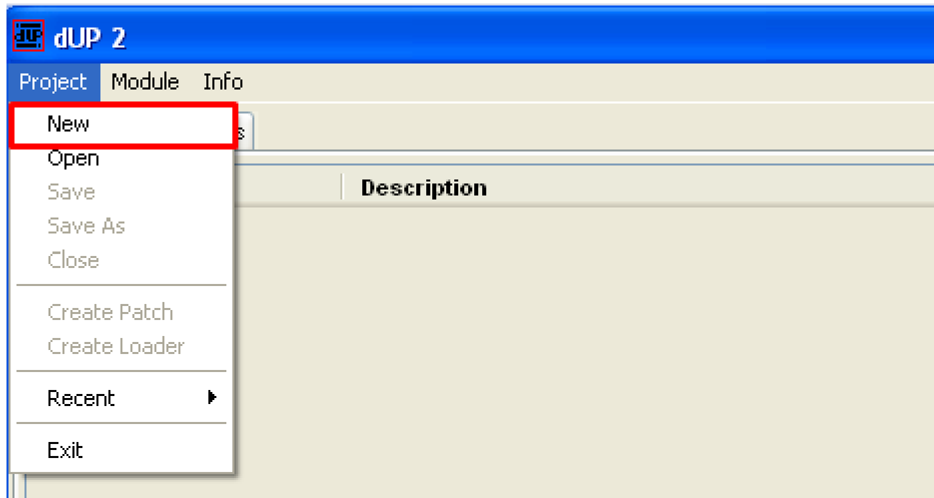
Una vez guardado el ejecutable ya modificado podríamos minimizar nuestro debugger e ir a buscar el parche y ver cómo nos ha quedado nuestro parche y si funciona correctamente ingresando cualquier clave

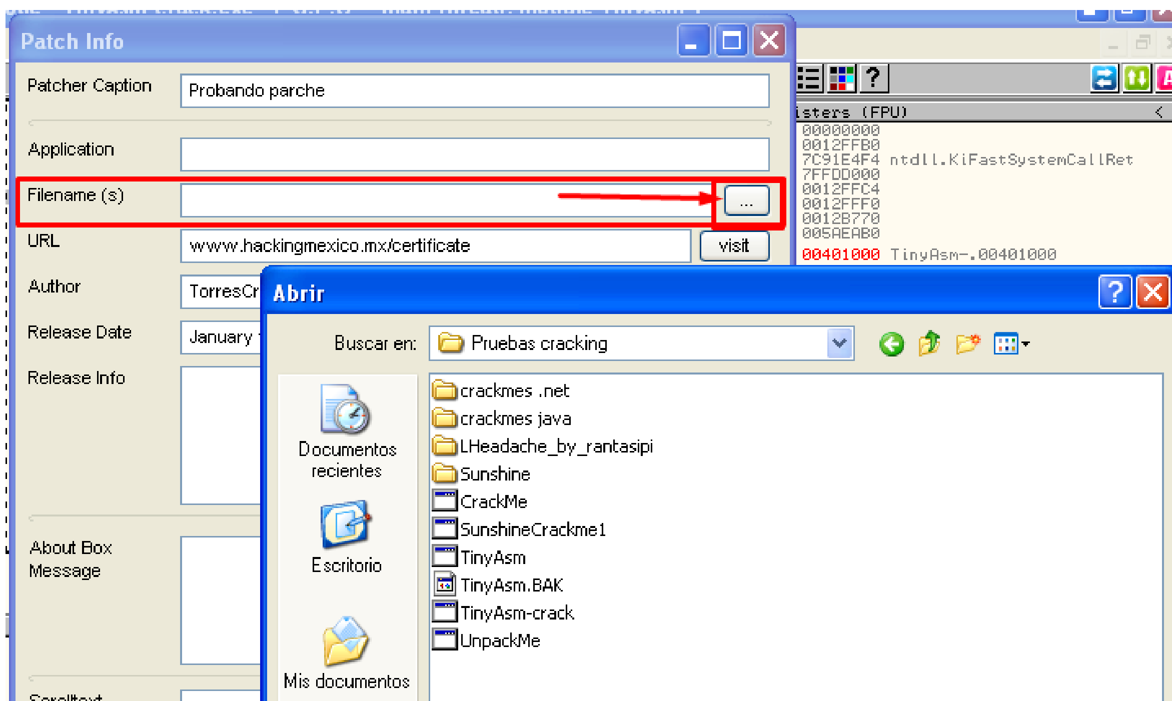


Ahora vamos a usar otra de nuestras herramientas para poder crear otra forma de crear un parche y es muy común encontrar por la red, un "parcheador" en este caso usaremos uno muy popular el "dup2", este tiene muchas opciones tanto para crear loader de diferente tipo y crearnos un parcheador, que basicamente se encargara únicamente de obtener los cambios entre el programa original y el parcheado y en base a esos cambios nos creara un ejecutable que sea capaz de parchear el ejecutable original tal como lo hicimos a mano pero este lo automatiza, pasemos a ejecutar la herramienta :

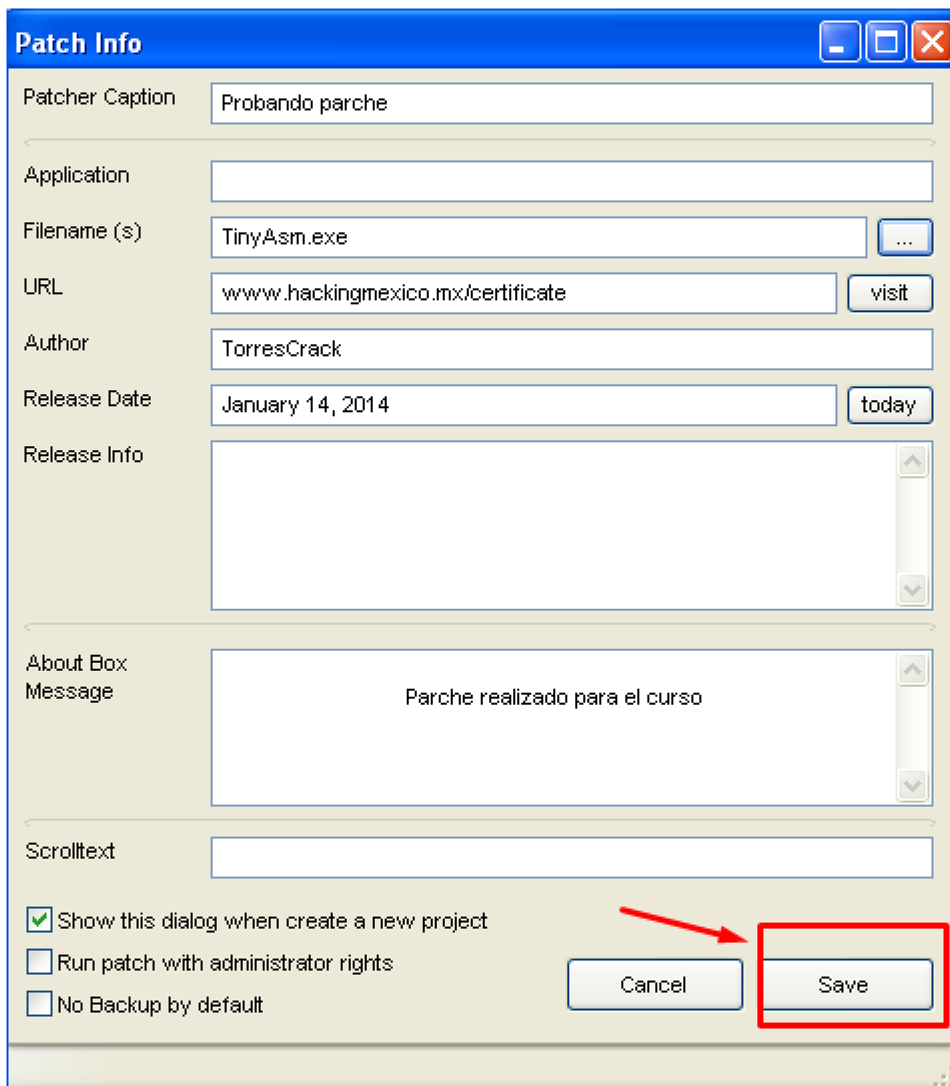


Una vez cargado seleccionamos en el menú "Project" y "NEW"

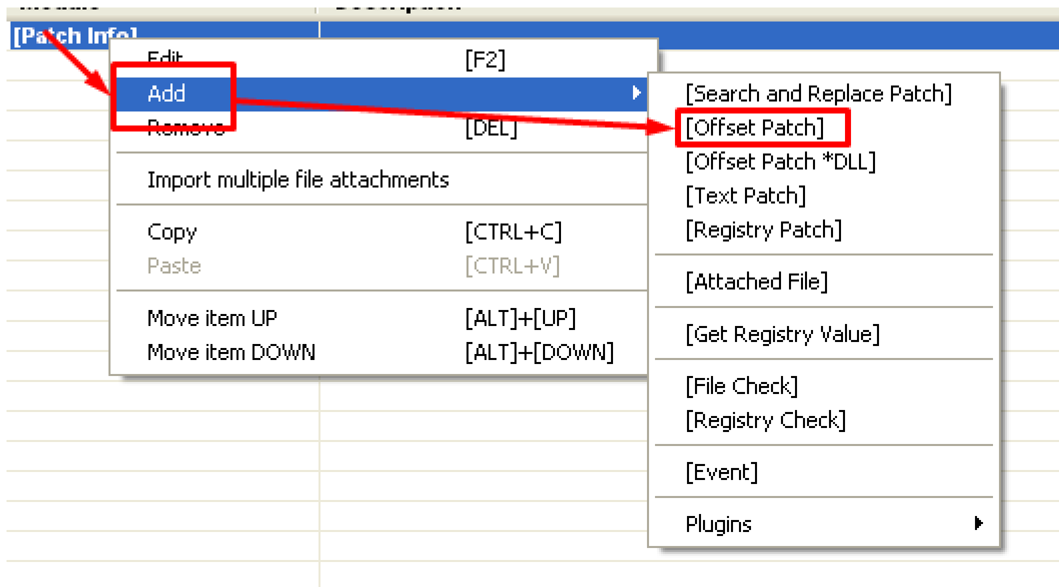


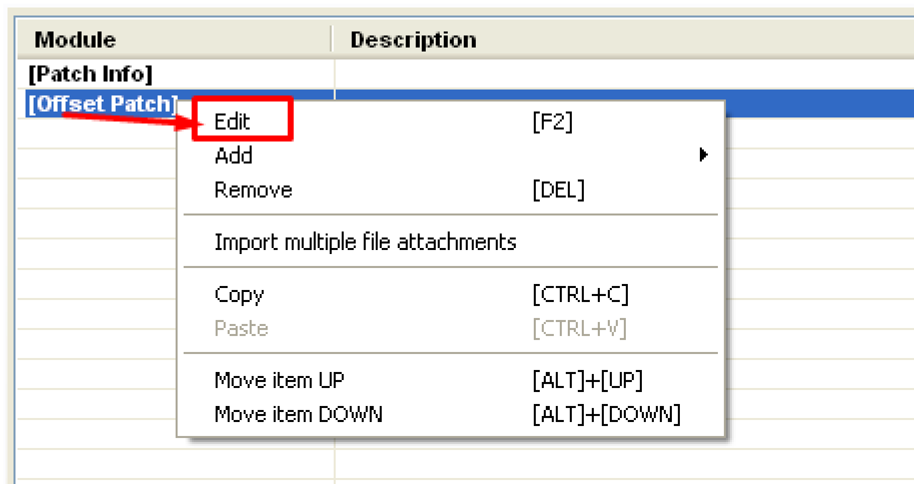


En la ventana que nos aparece, tiene varias casillas de texto para rellenar muchos datos hay una que nos dice "File Name" presionamos en el botón y seleccionamos nuestro programa ya quedando así todos los datos (que no es obligatorio rellenar)

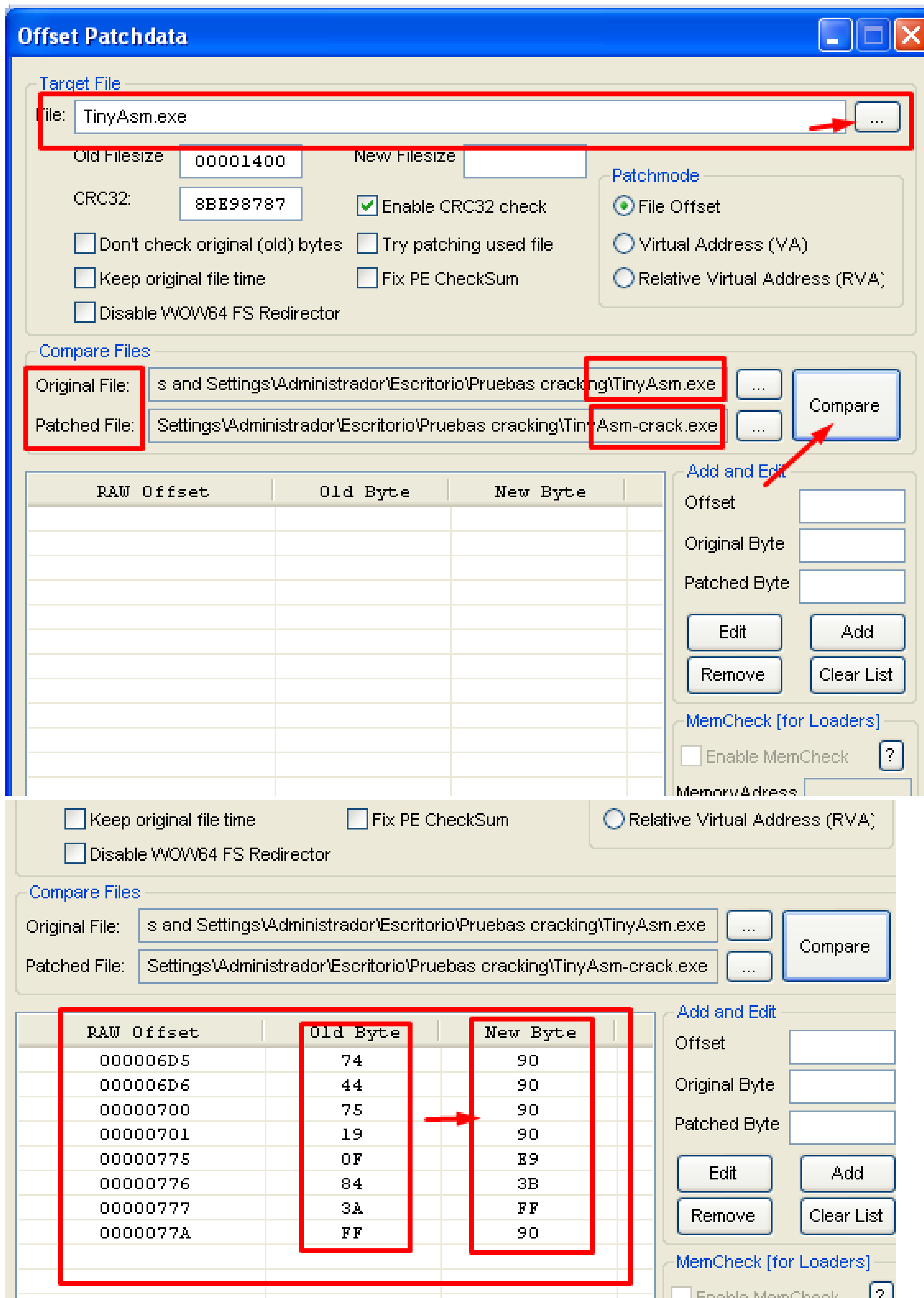


Presionamos el botón "save" y regresaremos a la ventana anterior hacemos click derecho en el "Patch info" y seleccionamos "add" y siguiente seleccionamos "offset patch".

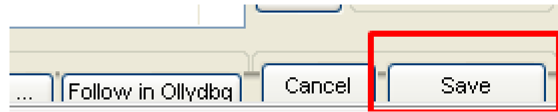




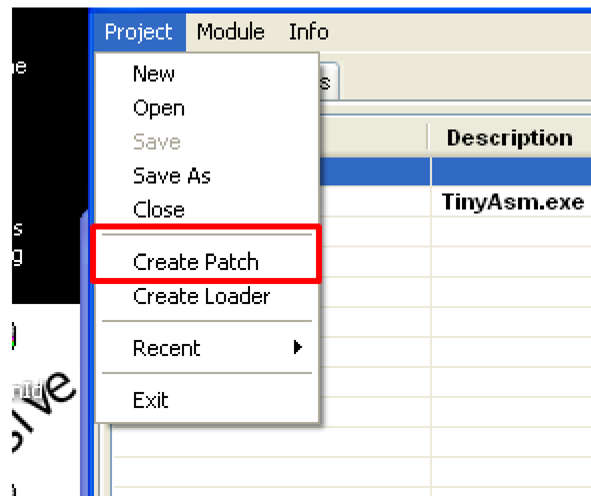
Ya creada la opción damos click derecho en ella y seleccionamos “edit” y veremos una ventana nueva donde debemos rellenar algunos datos, como lo es seleccionar a la víctima, seleccionar el programa original, el parcheado y en base a eso al presionar el botón “compare” hará lo correspondiente para ver qué cambios dentro del ejecutable hacen la diferencia, ya una vez cargado todos los datos presionamos el botón y veamos:

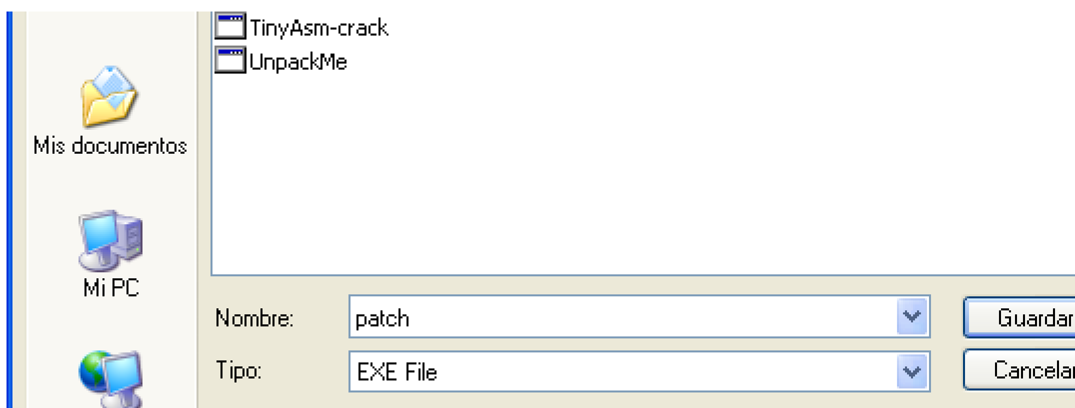

Offset Patchdata
Target File
File: TinyAsm.exe
Old Filesize 00001400
New Filesize
CRC32: 8BE98787
Enable CRC32 check
Don't check original (old) bytes
Try patching used file
Keep original file time
Fix PE CheckSum
Disable WOW64 FS Redirector
Patchmode
File Offset
Virtual Address (VA)
Relative Virtual Address (RVA)
Compare Files
Original File: s and Settings\Administrador\Escritorio\Pruebas cracking\TinyAsm.exe
Patched File: Settings\Administrador\Escritorio\Pruebas cracking\TinyAsm-crack.exe
Compare
RAW Offset
Old Byte
New Byte
Add and Edit
Offset
Original Byte
Patched Byte
Edit
Add
Remove
Clear List
MemCheck [for Loaders]
Enable MemCheck
000006D5 74 90
000006D6 44 90
00000700 75 90
00000701 19 90
00000775 0F E9
00000776 84 3B
00000777 3A FF
0000077A FF 90

Nos está mostrando los bytes originales del lado izquierdo y los parcheados o nuevos de lado derecho, ahora guardamos todo, con el botón “save” que esta hasta abajo:



Ahora en el menú seleccionamos “Project” seguidamente “CreatePatch” y después elegiremos en que ruta y como guardar nuestro parche:





Una vez guardado en la misma carpeta, podremos correrlo y con tan solo presionar el botón Patch este nos parchea el ejecutable sin ningún problema, este tipo de parcheadores es muy común y lo más común al descargarse un parche de internet ya que lo que facilita es el almacenamiento, en lugar de trasladar todo el parche del software , se transfiere este parcheador que pesa unos cuantos KB, ahora vayamos a la practica con un software real.

PRACTICANDO…