Ingeniería inversa con Internet Download Manager

PARTE I

Autor : Jmissing87
Contacto : Jmissing87@gmail.com
www.twitter.com/Jmissing87

Software : Internet Download Manager
Version : 6.23
S.O : Windows XP sp3

Índice :

Introducción _ :

El siguiente texto se ha escrito con la intención de informar y enseñar los distintos métodos que utilizan hoy en día los desarrolladores de software con la finalidad de proteger éste, si bien es un texto que describe el cómo funciona y el cómo se puede llegar a transgredir el método que se utiliza para validar a los usuarios que pagan por utilizar el software, la intención del creador del texto no es cometer algún acto delictivo y mucho menos incitar a un tercero.

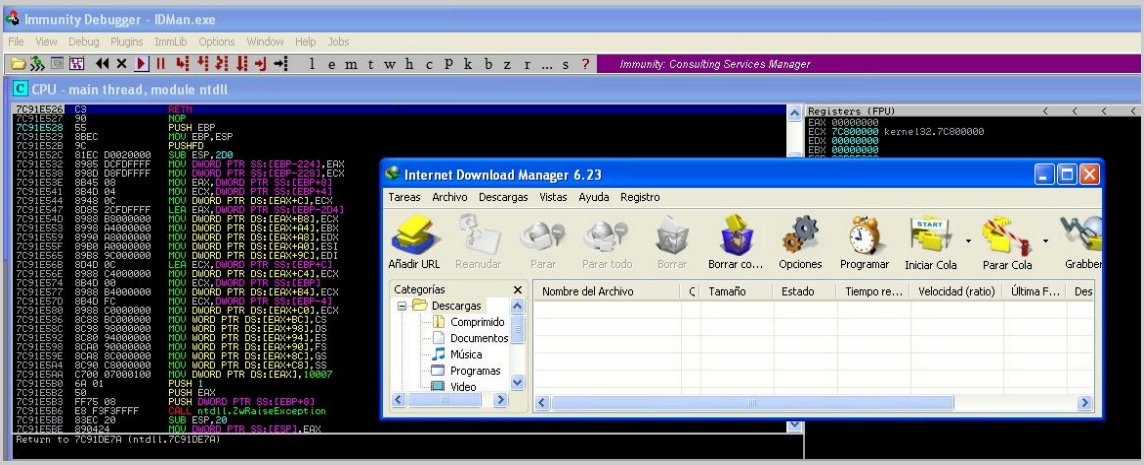
Dejándonos de bromas, soy autodidacta he dedicado bastante tiempo a aprender por mi mismo pero nunca nada me había costado tanto como aprender lo que es ingeniería inversa, pero gracias a la comunidad que han armado los chicos de *CRACKSLATINOS* y al genio de Ricardo Narvaja, he logrado comprender lo que significa ésto.

Dejo en claro que éste es mi primer texto que escribo soy un aprendiz del ***AIREINEGNI*** he disfrutado bastante haciéndolo y espero que ustedes también lo hagan leyéndolo, he dejado mi correo y twitter por si a alguien le ha quedado alguna duda o tiene una crítica.

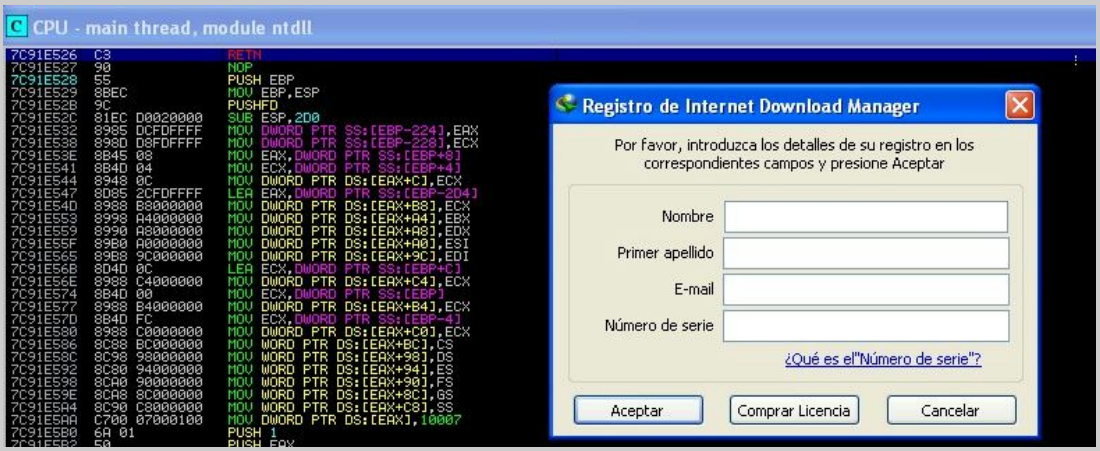
Cargando IDM :

Antes de comenzar a analizar IDM necesitamos cerrar el proceso, si tienes alguna descarga la pausamos, y cerremos desde la barra que aparece al lado del reloj, ***boton derecho -> salir*** ;) .

Una vez que cerremos abrimos ***immunity*** o ***ollydbg***, da igual, en mi caso utilizaré ***immunity*** solo por comodidad, abrimos ***IDM*** con ***immunity*** y presionamos ***F9***.



Vamos al botón de registrar y nos debe aparecer la siguiente pantalla.



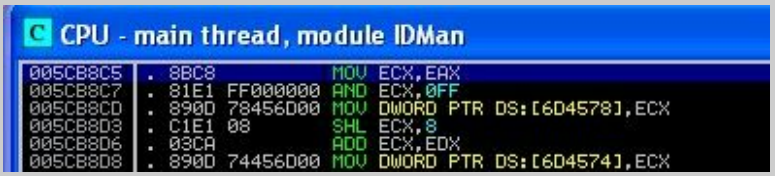
En ésta pantalla aparecen ***4*** campos, ***3*** de ellos que no son importantes y podemos rellenarlos con datos random, ésto porque el algoritmo que valida el serial no toma en cuenta ni uno de estos ***3*** campos, debemos prestar atención al ***4to*** campo, pero por ésta vez supondremos que no conocemos el formato ni los caracteres aceptados por el validador y rellenaremos con datos random también, solo por ésta vez ya que de apoco iremos descubriendo el formato del serial.

Antes de presionar el botón aceptar debemos seguir la pista del serial, planteare las dos formas que se me ocurrieron pero con la primera no llegue a buen puerto, perdí la pista de los datos ingresados por el usuario.

OJO *!*

Al presionar *F9* para que el programa se ejecute dentro del contexto de *immunity*, éste nos deja parado en la dll ***NTDLL***.

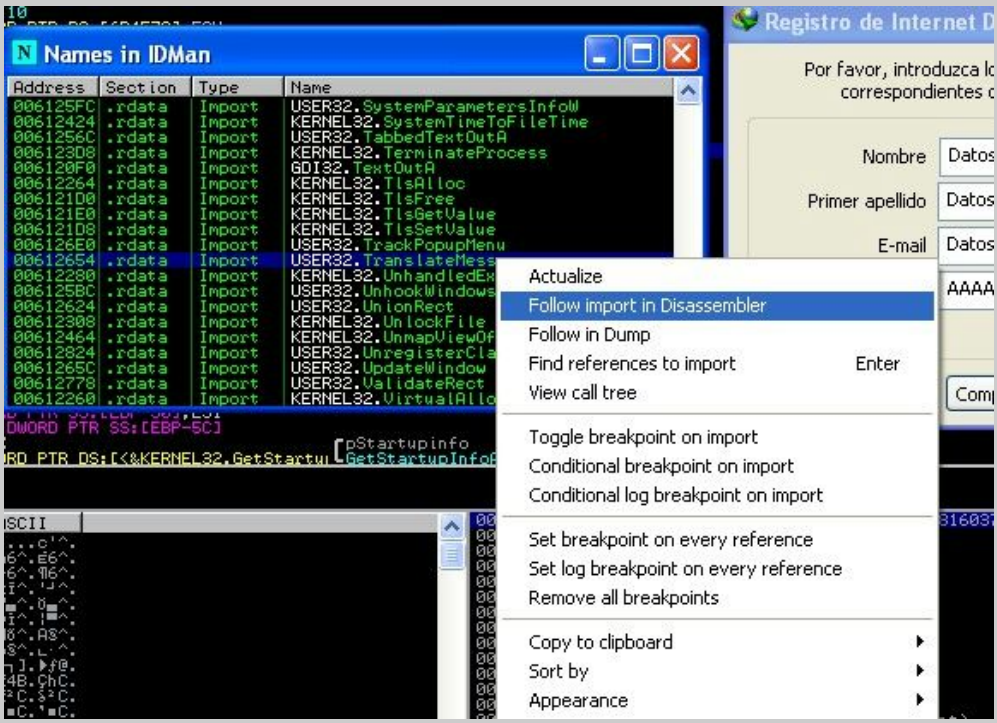
Por ésto debemos presionar el botón **'-'** para volver al módulo principal que corresponde al de ***IDM*** y continuar con el tutorial.



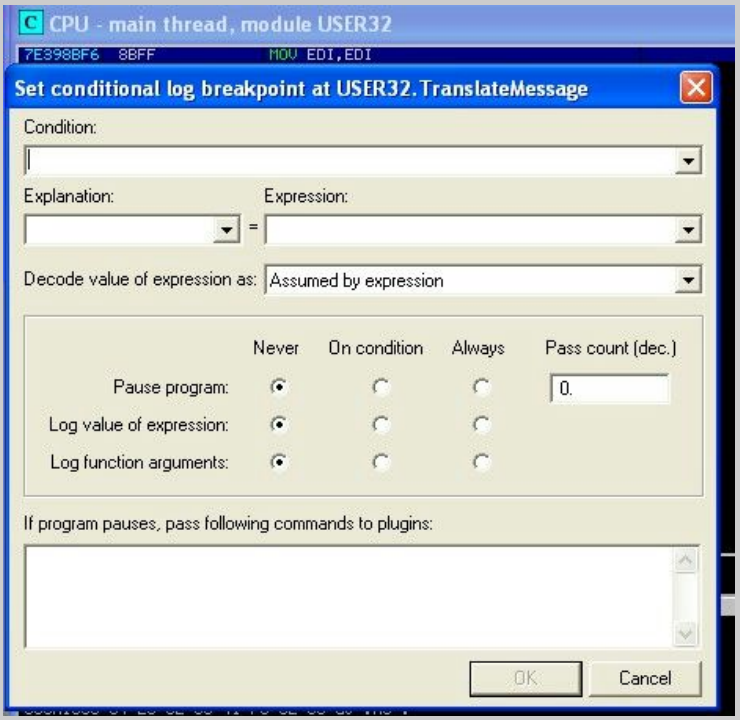
| *Cazando el serial 1* |

La primera manera que se me ocurrió fue poner un breakpoint "***Conditional Log***", ésto con la finalidad de tomar un api como ***TranslateMessage*** y capturar el evento 201 o 202(cuando se presiona o suelta el botón del mouse) del botón Aceptar, vamos a ver como se hace ésto.

Presionamos ***CTRL+N*** , y nos aparecerá la lista de Apis utilizada por ***IDM***, presionamos en un api y tipeamos ***TranslateMessage*** , en el recuadro nos aparecerá el api, presionamos botón derecho y "***Follow Import in Disassembler***".



Ahora nos encontramos parados en el inicio del api ***TranslateMessage***, si nuestro objetivo es poner un "***BP condicional Log***" éste es el lugar idóneo ya que podemos obtener fácilmente el valor de los parámetros y caer justo en el momento que se presione el botón aceptar.



Antes de poner el bp, debemos conseguir el ***HANDLE*** o el ***IDENTIFICADOR*** del botón "***aceptar***" y la información del API ***TranslateMessage*** para saber qué datos se pasan por parámetros y cómo extraerlos de manera correcta.

a.- *El handle del botón "**aceptar**" lo obtenemos desde el botón **[W]** de immunity.*

b.- *¿Qué parámetros recibe la función **TranslateMessage**?.*

//Información del API TranslateMessage

```
BOOL TranslateMessage(
    CONST MSG  *lpmsg   // address of structure with message
```

```
);
```

El api nos dice que recibe un puntero a una estructura de tipo ***MSG***, veamos los campos que contiene ésta estructura.

```
typedef struct tagMSG {     // msg
    HWND   hwnd;
    UINT   message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD  time;
    POINT  pt;
} MSG;
```

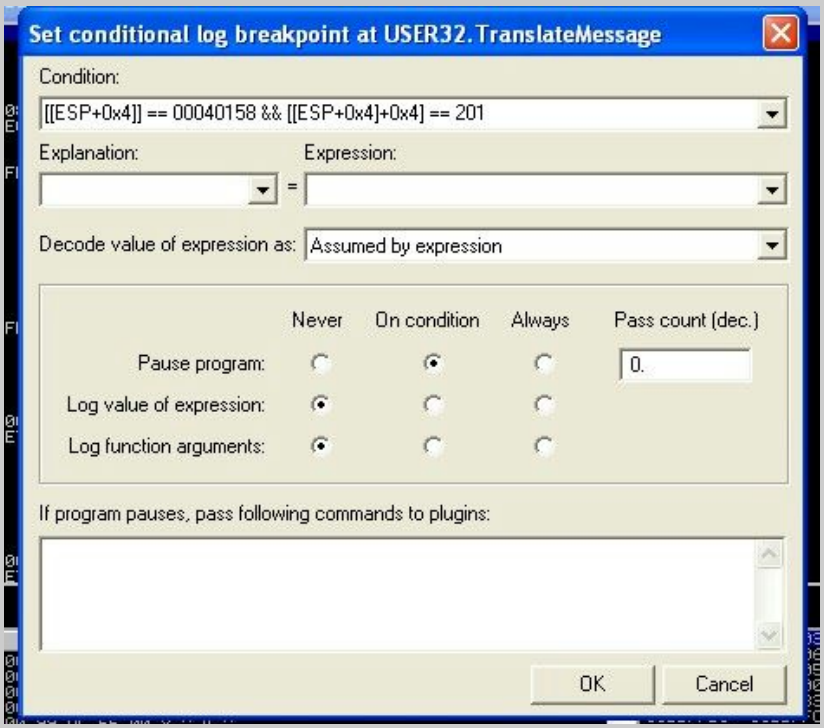
La estructura ***MSG*** se compone de 6 campos de los cuales utilizaremos el campo número 1 y 2 (***hwnd*** y ***message***) para filtrar.

c.- *Creamos el filtro necesario para parar nuestro programa.*

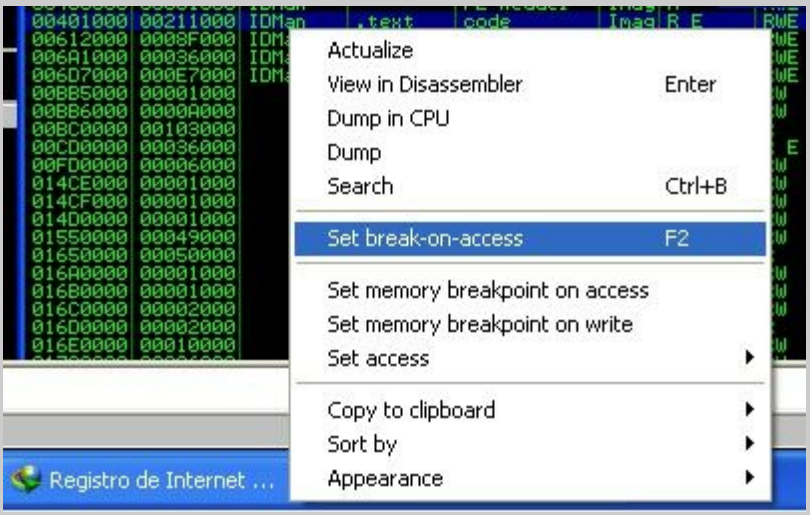
```
[[esp+0x4]] == Identificador_Boton && [[esp+0x4]+0x4] == 201
```

Donde:

[esp+0x4] => *Es el puntero a la estructura **MSG***
[[esp+0x4]] => *Es el valor del primer miembro de la estructura **MSG** (El handle del botón '**HWND**')*
[[esp+0x4]+0x4] => *Es el valor del segundo miembro de la estructura **MSG** (El tipo de Mensaje 'message')*



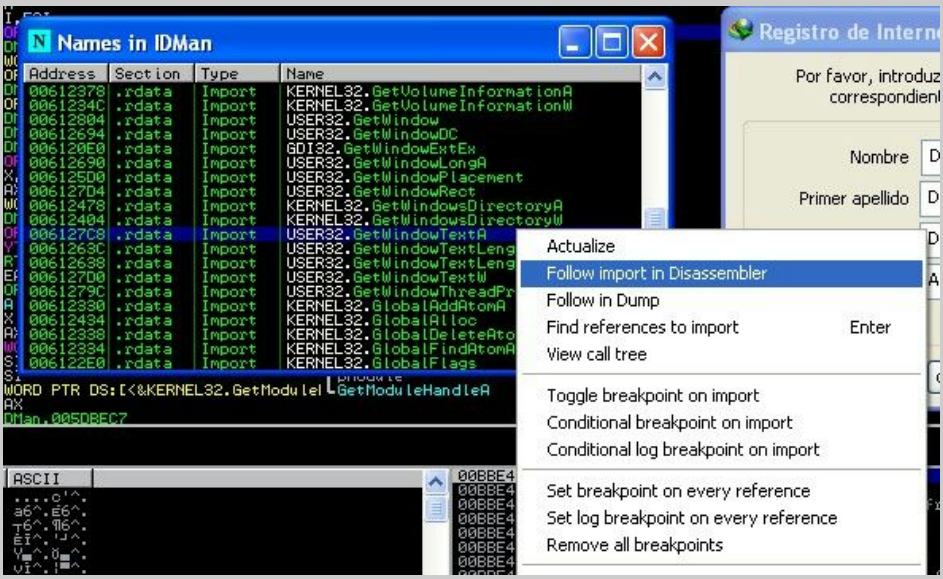
Una vez configurado el ***BP***, presionamos aceptar, el programa parará en la api ***TranslateMessage*** justo antes de procesar el mensaje, ahora debemos de alguna manera llegar a la sección de código donde se está ejecutando el programa, es decir al módulo principal. Para ello vamos al botón ***[M]*** de immunity y buscamos la sección ***.text*** del módulo ***IDMan*** ponemos un '***Break-On-Access***' y presionamos ***F9***, parará justo en el módulo de ***IDM***, y hasta aquí nomás llegué **:'(**, a mi manera de ver debía seguir



ejecutando una pila de código y sería demasiado tedioso continuar así .

| *Cazando el serial 2* |

Como no llegamos a buen puerto con la primera forma, lo realizaremos de la siguiente manera, debemos reiniciar immunity, una vez que tenemos el programa corriendo, buscamos las apis utilizadas por el programa ***CTRL+N*** , en el listado de apis buscamos ***GetWindowTextA*** presionamos en "***Follow Import in Disassembler***".

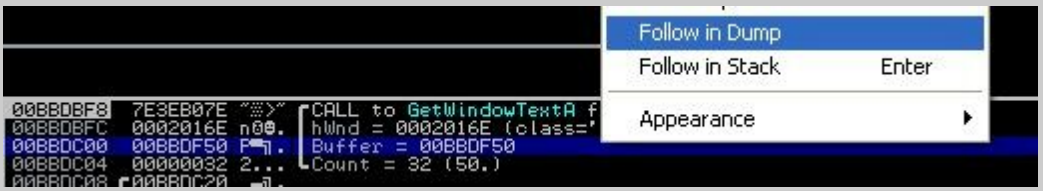


//Información del API GetWindowText

```
int GetWindowText(
    HWND  hWnd,          // handle of window or control with text
```

```
    LPTSTR  lpString,    // address of buffer for text
    int   nMaxCount      // maximum number of characters to copy
   );
```

Ponemos un bp con ***F2*** , rellenamos los campos de registro y presionamos aceptar, recordemos que los campos a rellenar son ***4*** por lo tanto el programa a lo menos debe parar ***4*** veces para obtener cada uno de estos datos, el ***4to*** breakpoint será el que tenga los datos del serial, por lo tanto al presionar aceptar ya paramos una vez en ***GetWindowTextA*** , debemos presionar ***F9*** 3 veces, con ésto paramos en la ***4ta*** llamada a ***GetWindowTexA*** y con ello conseguimos la dirección del buffer donde se almacena el serial.



dir. Serial : ***00BBDF50***

Miramos el stack obtenemos el ***2do*** parámetro(***Buffer***) que contiene la dirección del buffer, botón derecho y ponemos "***Follow in Dump***", presionamos ***CTRL+F9*** para que se ejecute la función y rellene el buffer con los datos del serial, luego en el dump tomamos los primeros ***4bytes*** y configuramos un "***Breakpoint -> Memory on acces***" , corremos el programa y ya estamos parados en el módulo principal y con la dirección del buffer :D , quitamos los breakpoint configurados y ahora intentaremos llegar al algoritmo de validación.

En éstos momentos nos encontramos parados en la instrucción.

004FFB33 |> 3895 7CFFFFFF /**CMP BYTE PTR SS:[EBP-84],DL**

Existe una comparación entre ***[ebp-84]*** y ***DL*** , si revisamos en el stack en la dirección ***[ebp-84]*** se encuentra nuestros serial y en ***DL*** podemos ver que se encuentra el contenido ***0x20*** que corresponde al carácter **' '**, es decir el programa está comprobando que el primer carácter del serial no sea un espacio.

```
$-88     > 00000000  ....
$-84     > 41414141  AAAA
$-80     > 41414141  AAAA
$-7C     > 42424141  AABB
$-78     > 42424242  BBBB
$-74     > 42424242  BBBB
$-70     > 00000000  ....
```

Continuamos ejecutando con F8 hasta después del salto condicional y llegamos a la instrucción .

```
004FFB97  |> 8DBD 7CFFFFFF  LEA EDI,DWORD PTR SS:[EBP-84]
```

Aquí vemos que nuevamente se accede a la dirección de ***[ebp-84]***, el programa está cargando el puntero al serial en el registro ***EDI*** .

En las siguientes instrucciones podemos ver un loop que se encarga de obtener el tamaño del serial, que se almacena en el registro ***ECX***, luego comprueba que la cantidad de caracteres no sea ***0***, de lo contrario nos mandará directamente fuera del programa diciendo que hemos introducido un serial incorrecto.

```
004FFB9D  |. 83C9 FF  OR ECX,FFFFFFFF
004FFBA0  |. 33C0     XOR EAX,EAX
004FFBA2  |. F2:AE    REPNE SCAS BYTE PTR ES:[EDI]
004FFBA4  |. F7D1     NOT ECX
004FFBA6  |. 49       DEC ECX
004FFBA7  |. 75 12    JNZ SHORT IDMan.004FFBBB
```

Seguimos ejecutando con ***F8*** hasta después del salto y nos topamos con otro loop similar.

```
004FFBBB  |> 8DBD 7CFFFFFF   /LEA EDI,DWORD PTR SS:[EBP-84]
004FFBC1  |. 83C9 FF         |OR ECX,FFFFFFFF
004FFBC4  |. 33C0            |XOR EAX,EAX
004FFBC6  |. F2:AE           |REPNE SCAS BYTE PTR ES:[EDI]
004FFBC8  |. F7D1            |NOT ECX
004FFBCA  |. 49              |DEC ECX
004FFBCB  |. 38940D 7BFFFFF> |CMP BYTE PTR SS:[EBP+ECX-85],DL
```

En ésta ocasión se encuentra comprobando que el contenido de la dirección ***[ebp + ecx - 85]*** no contenga el carácter de espacio, recordemos que en la dirección ***[ebp - 84]*** se encuentra el puntero a nuestro serial, la instrucción anterior accede a ***[ebp - 85]*** que corresponde a un carácter antes de nuestro serial y luego a esa dirección le suma el contenido del registro ***ECX*** que corresponde a la cantidad de caracteres que contiene nuestro serial.

```
00BBDF4B                  41 41 41  AAA
00BBDF53  41 41 41 41 41 41 41 62  AAAAAAAb
00BBDF5B  62 62 62 62 62 62 62 62  bbbbbbbb
00BBDF63  62                       b
```

Si realizamos el cálculo nos dará como resultado ***00BBDF63*** que contiene el último carácter de nuestro serial, por lo tanto ya sabemos que nuestro serial no puede comenzar por un carácter de espacio y tampoco terminar con uno de éstos.

Seguimos ejecutando el programa con ***F8*** y llegamos a un call que recibe como parámetro la dirección del buffer donde se almacena nuestro serial, entraremos a ese ***CALL*** con ***F7*** .

```
004FFBED  |> 8D95 7CFFFFFF  LEA EDX,DWORD PTR SS:[EBP-84]
004FFBF3  |. 52              PUSH EDX
004FFBF4  |. E8 92A90C00     CALL IDMan.005CA58B
```

Dentro del ***CALL*** las líneas que nos importan son las siguientes.

```
005CA59F  |. 8B45 08        MOV EAX,DWORD PTR SS:[EBP+8]
005CA5A2  |. 8BD0           MOV EDX,EAX
005CA5A4  |. 8038 00        CMP BYTE PTR DS:[EAX],0
```

Obtenemos la dirección del serial y la almacenamos en ***EAX***, luego la movemos a ***EDX*** para posteriormente comprobar con el ***CMP*** si el contenido de esa dirección es un ***NULL BYTE*** ! si es así nos saca de la función, de lo contrario nos manda a un loop .

```
005CA5AD  |> 8A0A          /MOV CL,BYTE PTR DS:[EDX]
005CA5AF  |. 80F9 61        |CMP CL,61
005CA5B2  |. 7C 0A          |JL SHORT IDMan.005CA5BE
005CA5B4  |. 80F9 7A        |CMP CL,7A
005CA5B7  |. 7F 05          |JG SHORT IDMan.005CA5BE
005CA5B9  |. 80E9 20        |SUB CL,20
005CA5BC  |. 880A          |MOV BYTE PTR DS:[EDX],CL
005CA5BE  |> 42            |INC EDX
005CA5BF  |. 803A 00        |CMP BYTE PTR DS:[EDX],0
005CA5C2  |.^75 E9          \JNZ SHORT IDMan.005CA5AD
```

En el loop vemos ***3*** constantes .

0x61 -> *'a'*
0x7A -> *'z'*
0x20 -> *Distancia entre 'A' y 'a' en la tabla ascii*

En simples palabras el loop se encarga de verificar si los caracteres ingresados en el serial estan en minusculas si lo están le resta ***0x20*** para pasarlo a mayúsculas, es toda la ciencia de ésta loop y la finalidad de éste ***CALL***.

Presionamos ***CTRL+F9*** y salimos de la función, al salir de ésta nos topamos con otro loop.

```
004FFBF9  |. 8DBD 7CFFFFFF  LEA EDI,DWORD PTR SS:[EBP-84]
004FFBFF  |. 83C9 FF        OR ECX,FFFFFFFF
004FFC02  |. 33C0           XOR EAX,EAX
004FFC04  |. 83C4 04        ADD ESP,4
004FFC07  |. F2:AE          REPNE SCAS BYTE PTR ES:[EDI]
004FFC09  |. F7D1           NOT ECX
004FFC0B  |. 49             DEC ECX
004FFC0C  |. 83F9 17        CMP ECX,17
```

Éste tipo de loop ya lo conocemos y nos hemos topado con él por lo menos ***2*** veces antes, el loop se encarga de obtener la cantidad de caracteres ingresados en el serial y almacena la cantidad en ***ECX***, al final del loop compara el valor de ***ECX*** con ***0x17*** que en decimal es ***23***, como mi serial consta solo de ***20*** caracteres el programa me dirá que he ingresado un serial incorrecto.

Guardamos la dirección antes de que el programa nos envíe el mensaje, en mi caso anoté la dirección del comienzo del loop.

```
004FFBF9  |. 8DBD 7CFFFFFF  LEA EDI,DWORD PTR SS:[EBP-84]
```

Reiniciamos el programa, ponemos un ***BP*** en la dirección que anotamos anteriormente, en mi caso ***004FFBF9***, ahora al ingresar los datos de registro debemos anotar un serial de ***23*** caracteres,con esto pasaremos a la siguiente etapa de validación.

```
004FFC11  |. 8A4D 81        MOV CL,BYTE PTR SS:[EBP-7F]
004FFC14  |. 8845 EF        MOV BYTE PTR SS:[EBP-11],AL
004FFC17  |. B0 2D          MOV AL,2D
004FFC19  |. 3AC8           CMP CL,AL
004FFC1B  |. 75 0A          JNZ SHORT IDMan.004FFC27
```

Sabemos que nuestro buffer con el serial se encuentra en la dirección ***[EBP-84]***, si vemos la primera línea después del loop vemos que mueve un byte de la dirección ***[EBP-7F]***, lo que hace es mover un ***BYTE*** del buffer con el serial que ingresamos.

N°	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
C	A	A	A	A	A	A	A	A	A	A	B	B	B	B	B	B	B	B	B	B	c	c	c
Ebp	84	83	82	81	80	7F	7E	7D	7C	7B	7A	79	78	77	76	75	74	73	72	71	70	6F	6E

El ***MOV*** está moviendo el carácter '***A***'(ubicado en la ***6ta*** posición) al registro ***CL***, la siguiente instrucción debemos tenerla en cuenta ya que el byte que se almacena en ***[EBP-11]*** es un ***FLAG*** y que se utilizará durante todo el algoritmo de validación por ahora está a '***0***' y esperamos que durante toda la validación sea así, ahora veamos que el valor ***0x2D*** corresponde al carácter '**-**', lo compara con nuestra '*A*', si no son iguales, continuará con la validación pero al final de todo nos dejará el ***FLAG [EBP-11]*** a '***1***' nos enviará a registrarnos con un serial correcto **:'(** .

Si miramos el código de más abajo, podemos deducir el formato del serial que debemos ingresar para pasar todas las validaciones sin activar el ***FLAG [EBP-11]***, fijémonos en las instrucciones siguientes.

```
004FFC1D   |. 3845 87      CMP BYTE PTR SS:[EBP-79],AL
004FFC20   |. 75 05        JNZ SHORT IDMan.004FFC27
004FFC22   |. 3845 8D      CMP BYTE PTR SS:[EBP-73],AL
004FFC25   |. 74 04        JE SHORT IDMan.004FFC2B
```

Vemos que compara ahora ***[EBP-79]*** y ***[EBP-73]*** debe tener el carácter '**-**', es decir cada ***5*** caracteres hay un carácter '**-**', el formato del serial debe ser así.

N°	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
C	A	A	A	A	A	-	B	B	B	B	B	-	c	c	c	c	c	-	d	d	d	d	d
Ebp	84	83	82	81	80	7F	7E	7D	7C	7B	7A	79	78	77	76	75	74	73	72	71	70	6F	6E

El serial :

a.- *Debe contener caracteres, da igual si ingresamos en mayusculas o minusculas, ya que el programa se encarga de pasarlo a mayúsculas.*

b.-*Se debe ingresar 4 grupos de 5 caracteres.*

c.- *Cada grupo debe ser separado por un '-' .*

d.- *Total : (**4**grupos x **5**caracteres) + **3**guiones = **23** o **0x17***

A pesar de que el flag se ha activado, continuemos con el análisis, más adelante veremos las consecuencias que acarrea dejar éste flag en ***1*** **;)** , si continuamos con las instrucciones, resaltan ***4 CALL's*** sospechosas, si sabemos que el serial se divide en '***4***' grupos de '***5***' caracteres, analicemos los parámetros de los ***CALL's*** .

```
004FFC2B  |> 8D85 7CFFFFFF      LEA EAX,DWORD PTR SS:[EBP-84]
004FFC31  |. 6A 05              PUSH 5
004FFC33  |. 8D4D D4            LEA ECX,DWORD PTR SS:[EBP-2C]
004FFC36  |. 50                 PUSH EAX
004FFC37  |. 51                 PUSH ECX
004FFC38  |. E8 A37B0C00        CALL IDMan.005C77E0
```

De antemano ya sabemos que en ***[EBP-84]*** se encuentra nuestro serial, luego en ***[EBP-2C]*** al acceder a la dirección que me entrega, puedo distinguir solo caracteres basura y deduzco que se puede tratar de un buffer.

[ebp-2c]

```
....
00BBDFA8  24 5D 55 01 00 00 00 00  $]U
```

Entonces podemos ver que los parámetros quedan de la siguiente forma.

EAX *= Puntero al serial ingresado por nosotros.*
ECX *= buffer.*
5 *= cantidad de caracteres que procesa la función.*

Podemos deducir que el 5 corresponde a la cantidad de caracteres, por el hecho de que en los siguientes **CALL's** se restan posiciones al buffer del serial con la finalidad de procesar los siguientes datos.

1 CALL => *DWORD PTR SS:[EBP-84] : "AAAAA"*

2 CALL => *DWORD PTR SS:[EBP-7E] : "AAAAB"*

3 CALL => *DWORD PTR SS:[EBP-78] : "BBBBB"*

4 CALL => *DWORD PTR SS:[EBP-72] : "BBCCC"*

Esos son los parámetros relacionados con el serial ingresado de los 4 **CALL's** , en cada dirección se van restando de **6** posiciones, ésto porque se evita introducir el carácter **'-' ;)** .

Ejecutaremos el primer **CALL** sin ingresar a él, posteriormente revisaremos qué sucede con el buffer que se ingresa por parámetro, pasamos con **F8** el **CALL** y vemos que en el buffer se han ingresado los primeros 5 caracteres, sigamos con los siguientes CALL's y veremos que con todos se hace lo mismo, es decir, esos CALL sirven solo para separar los 4 grupos de 5 caracteres en buffer distintos.

```
004FFC6A  |. 33FF         XOR EDI,EDI
004FFC6C  |. 83C4 30      ADD ESP,30
004FFC6F  |. C645 D9 00   MOV BYTE PTR SS:[EBP-27],0
004FFC73  |. C645 D1 00   MOV BYTE PTR SS:[EBP-2F],0
004FFC77  |. C645 E5 00   MOV BYTE PTR SS:[EBP-1B],0
004FFC7B  |. C645 B5 00   MOV BYTE PTR SS:[EBP-4B],0
004FFC7F  |. 897D E8      MOV DWORD PTR SS:[EBP-18],EDI
```

Saliendo ya de los CALL's podemos ver como al final de cada buffer se añade un **NULL BYTE**, en este caso aparecen 5 buffer, el 5to que es ***[ebp-18]*** se utilizará más abajo y ya lo veremos, por ahora recordemos que tenemos un flag con valor ***'1'*** en ***[ebp-11]*** y el buffer ***[EBP-18]***.

Continuando con el análisis, en las siguientes 3 instrucciones podemos deducir que se nos viene un loop.

```
004FFC82  |. 33F6      XOR ESI,ESI
004FFC84  |> 83FE 05   CMP ESI,5
004FFC87  |. 7D 32     JGE SHORT IDMan.004FFCBB
```

Analizando el Loop podemos observar que una de las condiciones será que **ESI** sea Menor a '5', también podemos ver que se obtiene el primer carácter del primer buffer y se almacena en ***DL***.

```
004FFC89  |. 8A5435 D4      MOV DL,BYTE PTR SS:[EBP+ESI-2C]
```

El buffer se encuentra en ***[EBP-2c]*** , ESI trabajará como contador para recorrer el buffer de 5 caracteres.

```
004FFC90  |. 33C0      XOR EAX,EAX
004FFC92  |> 83F8 24   /CMP EAX,24
004FFC95  |. 7D 0A     |JGE SHORT IDMan.004FFCA1
```

Podemos ver que EAX se limpia y otra condición del loop, la condición es que EAX sea menor o igual a 0x24 es decir 36 en decimal, pero ***¿ ese 0x24 a que se debe ?***, en la siguiente línea encontramos la respuesta.

```
004FFC97  |. 3890 A8156D00  |CMP BYTE PTR DS:[EAX+6D15A8],DL
```

Veamos que hay en la dirección "**006D15A8**".

```
006D15A8  32 59 4F 50 42 33 41 51  2YOPB3AQ
006D15B0  43 56 55 58 4D 4E 52 53  CVUXMNRS
006D15B8  39 37 57 45 30 49 5A 44  97WE0IZD
006D15C0  34 4B 4C 46 47 48 4A 38  4KLFGHJ8
006D15C8  31 36 35 54 2C           165T,
```

Vemos que hay una tablita con letras de la '***A***' a la '***Z***' y con los números del ***1*** al ***9***, veamos cómo se relaciona esta tabla con el buffer que contiene los primeros ***5*** caracteres del serial.

```
004FFC97  |. 3890 A8156D00  |CMP BYTE PTR DS:[EAX+6D15A8],DL
004FFC9D  |. 75 15          |JNZ SHORT IDMan.004FFCB4
```

La comparación anterior se realiza entre nuestro primer carácter del buffer(***ESI***=*0*) con los ***0x24*** o ***36*** caracteres de la tabla que encontramos, sí el carácter del serial no corresponde se aumentará el contador de la tabla(***EAX***=*0*, ***EAX***=*1*, ***EAX*** ...), es decir cada carácter de nuestro serial es comparado en esa tabla hasta que coincide con un valor de la tabla, pero ***¿ qué pasa cuando uno de esos caracteres coinciden ?*** .

```
004FFC9F  |. 8BC8       |MOV ECX,EAX
004FFCA1  |> 83F9 FF    |CMP ECX,-1
004FFCA4  |. 74 11      |JE SHORT IDMan.004FFCB7
004FFCA6  |. 8D14FF     |LEA EDX,DWORD PTR DS:[EDI+EDI*8]
004FFCA9  |. 03CF       |ADD ECX,EDI
004FFCAB  |. 46         |INC ESI
004FFCAC  |. 8D3C91     |LEA EDI,DWORD PTR DS:[ECX+EDX*4]
004FFCAF  |. 897D E8    |MOV DWORD PTR SS:[EBP-18],EDI
004FFCB2  |.^EB D0      |JMP SHORT IDMan.004FFC84
```

Se copia el **VALOR DEL CONTADOR EAX(CORRESPONDE AL DE LA TABLA) A ECX, LUEGO ÉSTE ES COMPARADO CON -1 DE SER IGUAL A -1 FIJENSE QUE SALTA A LA DIRECCIÓN** "***004FFCB7***" si salta a esa dirección nos encontramos con.

```
004FFCB7  |> C645 EF 01     MOV BYTE PTR SS:[EBP-11],1
```

El ya mencionado ***FLAG [EBP-11]***, se setea a ***1*** el ***FLAG***, recordemos que nosotros ya tenemos el ***FLAG*** a ***1*** y mencioné que por ahora éste no nos dará problemas pero si al final .

Siguiendo con el análisis, si ***EAX*** o ***ECX*** no conservan el valor ***-1***, ocurre que se realiza un serie de cálculos, donde ***EDI*** será el registro que acumula los valores calculados, no le debemos perder la ***PISTA***.

```
004FFCA6  |. 8D14FF   |LEA EDX,DWORD PTR DS:[EDI+EDI*8]
004FFCA9  |. 03CF     |ADD ECX,EDI
004FFCAB  |. 46       |INC ESI
004FFCAC  |. 8D3C91   |LEA EDI,DWORD PTR DS:[ECX+EDX*4]
004FFCAF  |. 897D E8  |MOV DWORD PTR SS:[EBP-18],EDI
```

Si pasamos ese algoritmo a algo un poco más legible como *'C'*, tenemos lo siguiente.

```
#include <stdio.h>
#include <string.h>

int main(int argc , char** argv)
{

//Tabla ubicada en [EAX+6D15A8] Donde EAX=0x0 hasta EAX<0x24.
char cadena[]       = "2YOPB3AQCVUXMNRS97WE0IZD4KLFGHJ8165T";

//Buffer ubicado en [EBP+ESI-2C] Donde ESI=0x0 hasta ESI<0x5.
char buff[]         = "AAAAA";

//Contador de la tabla llegará hasta EAX<0x24.
int eax = 0;
//Contador Del Buffer llegará hasta ESI<0x5.
int esi = 0;
//Registro acumulador, nos dará el resultado final de la operación.
int edi = 0;
//Se utilizan como variables temporales y ayuda a realizar cálculos.
int ecx = 0;
int edx = 0;


for(esi=0; esi < 5; esi++)
{
     for(eax=0; eax < strlen(cadena); eax++ )
     {

          if( cadena[eax] == buff[esi] )
          {
               printf("cadena[%d]=%c  ==
buff[%d]=%c\n",eax,cadena[eax],esi,buff[esi]);
               ecx = eax;
               edx = edi + edi * 8;
               ecx = ecx + edi;
               edi = ecx + edx * 4;
               printf("edi %d\n",edi);
          }
     }
}

    printf("final 0x%08x \n",edi);

    return 0;
}
```

Éste algoritmo se utilizará en los 3 loops restantes .

```
1.- 004FFCD0  |> 83F8 24    /CMP EAX,24
2.- 004FFD0B  |> 83F8 24    /CMP EAX,24
3.- 004FFD43  |> 83F8 24    /CMP EAX,24
```

Los loops trabajan de la misma forma, por lo tanto solo nos interesará obtener los resultados, además debemos anotar las direcciones donde se almacenan los resultados del algoritmo.

```
1er loop : 004FFCAF  |. 897D E8    |MOV DWORD PTR SS:[EBP-18],EDI
2do loop : 004FFCED  |. 897D DC    |MOV DWORD PTR SS:[EBP-24],EDI
3er loop : 004FFD25  |. 8D1C91     |LEA EBX,DWORD PTR DS:[ECX+EDX*4]
4to loop : 004FFD5D  |. 8D3C81     |LEA EDI,DWORD PTR DS:[ECX+EAX*4]
```

Sabemos que la direcciones que debemos seguir son :

Dirección		Contenido
00BBDFBC	[EBP-18]	00B059CE
00BBDFB0	[EBP-24]	00B059CC
EBX		00759134
EDI		0075A730

Tenemos que de los buffer siguientes se calcularon los valores :

```
'AAAAA' -> 00B059CE
'AAAAB' -> 00B059CC
'BBBBB' -> 00759134
'BBCCC' -> 0075A730
```

Al salir de los 4 loops nos encontramos con el siguiente problema y que ya les había mencionado.

```
004FFD65  |> 8A45 EF    MOV AL,BYTE PTR SS:[EBP-11]
004FFD68  |. 84C0       TEST AL,AL
004FFD6A  |. 74 16      JE SHORT IDMan.004FFD82
```

Les había mencionado que en la dirección ***[EBP-11]*** se encuentra un byte que ***IDM*** lo utiliza como ***FLAG***, también les había mencionado que nuestro serial al no tener el formato '***AAAAA-BBBBB-CCCCC-DDDDD***' ya se había activado el ***FLAG***, por lo tanto llegó el momento en el que ***IDM*** comprueba si este ***FLAG*** se encuentra en ***1*** o en ***0***, si el flag se encontrara en ***0*** podríamos seguir validando los datos pero lamentablemente ya se configuró en ***1*** , de todos modos para continuar con nuestro análisis modificaremos ese valor.

Seguimos recorriendo ***IDM*** hasta después del salto condicional y vemos donde caemos.

```
004FFD82   |> 8B4D E8         MOV ECX,DWORD PTR SS:[EBP-18]
004FFD85   |. BE 2B000000     MOV ESI,2B
004FFD8A   |. 8BC1            MOV EAX,ECX
004FFD8C   |. 99              CDQ
004FFD8D   |. F7FE            IDIV ESI
004FFD8F   |. 85D2            TEST EDX,EDX
004FFD91   |. 75 04           JNZ SHORT IDMan.004FFD97
```

El ***MOV*** toma los ***4bytes*** de la dirección ***EBP-18***,si vamos a esa dirección veremos que el contenido de ésta es '***00B059CE***' que coincide con el valor que retornó el algoritmo cuando procesaba el contenido del primer buffer es decir '***AAAAA***', continuamos revisando las instrucciones y podemos ver que en ***ESI*** se mueve una constante ***0x2B*** que en decimal es ***43***, abajo nuevamente carga el cálculo del algoritmo de ***ECX*** a ***EAX*** y luego se realiza una división con el valor que se encuentra en ***ESI***.

```
EAX : ESI = ESI    RESTO : EDX
```

Tomando los valores correspondientes.

```
0xB059CE : 0x2b = 0x419e7    RESTO : 0x1
```

El algoritmo comprueba si el resto de la división es ***0x0***, en nuestro caso no será ***0x0***, como vimos el resultado de la división es ***0x419e7*** y el resto es ***0x1***.

Si el resto es 0x1 miremos donde salta.

```
004FFDB0  |> C645 EF 01   MOV BYTE PTR SS:[EBP-11],1
```

Nuevamente el flag que se encuentra en ***[EBP-11]*** será activado **:'(** .

De todos modos podemos seguir analizando los siguientes loop y veremos que cambian 2 parámetros, el primero es el resultado del primer algoritmo y el segundo es la constante por la cual se divide, tomemos nota.

```
1er loop -> Resultado1 = 00B059CE   ; Constante = 0x2b
2do loop -> Resultado1 = 00B059CC   ; Constante = 0x17
3er loop -> Resultado1 = 00759134   ; Constante = 0x11
4to loop -> Resultado1 = 0075A730   ; Constante = 0x35
```

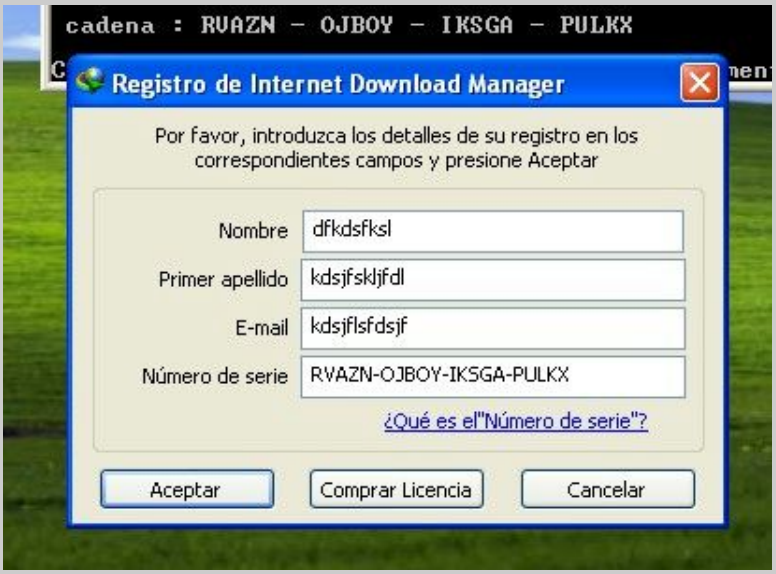
Para aclarar, necesitamos un buffer que al pasar por el algoritmo que escribimos en ***C*** nos de un número múltiplo de ***0x2b*** para que cuando se divida por ***0x2b*** nos entregue como ***RESTO 0x0***, el segundo buffer debe ser múltiplo de ***0x17,*** el tercero de ***0x11*** y el cuarto de ***0x35*** .

Pasando ésta validación nos encontraremos con ***IDM*** registrado, necesitamos generar el serial con los parámetros anteriormente mencionados y para ello lo haremos de la manera menos recomendada pero para éste caso nos sirve, lo realizaremos con fuerza bruta ***:P*** .

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

//Corresponde a la letra 'A' y 'Z' en ASCII
int INICIO=65;// 'A'
int FIN=90;   // 'Z'

//genera serial random
char* generar( int );
```

```
int main(int argc , char** argv)
{
     srand(time(NULL));
     printf("\nSerial     :     %s     -     %s     -     %s     -     %s\n",
generar(0x2b),generar(0x17),generar(0x11),generar(0x35));

    return 0;
}


char* generar( int div )
{
      //Tabla ubicada en [EAX+6D15A8] Donde EAX=0x0 hasta EAX<0x24.
      char cadena[]       = "2YOPB3AQCVUXMNRS97WE0IZD4KLFGHJ8165T";
      //Buffer ubicado en [EBP+ESI-2C] Donde ESI=0x0 hasta ESI<0x5.
      char buff[5]         = {'\0'};
      //Contador de la tabla llegará hasta EAX<0x24.
      int eax = 0;
      //Contador Del Buffer llegará hasta ESI<0x5.
      int esi = 0;
      //Registro acumulador, nos dará el resultado final de la operación.
      int edi = 0;
      //Se utilizan como variables temporales y ayuda a realizar cálculos.
      int ecx = 0;
      int edx = 0;


      for(int i=0; i<5; i++)
      {
          buff[i] = (char)(rand() % (FIN - INICIO + 1) + INICIO );
      }

      for(esi=0; esi < 5; esi++)
      {
          for(eax=0; eax < strlen(cadena); eax++ )
          {
               if( cadena[eax] == buff[esi] )
               {
                    ecx = eax;
                    edx = edi + edi * 8;
                    ecx = ecx + edi;
                    edi = ecx + edx * 4;
               }
          }
      }

      if((edi%div)==0)
      {
          char* tmp=(char*)malloc(sizeof(char)*6);
          memset(tmp,0,sizeof(char)*6);
```

```
            strncpy(tmp,buff,5);
            return tmp;
        }else{ return generar(div); }
}
```



Como vemos podemos registrar el programa sin problemas, ya que EL SERIAL ha pasado TODOS los filtros, el problema viene después ya que IDM realiza un validación a través de sus servidores de internet por lo tanto veremos un mensaje como el siguiente.

La solución a éste problema ***ERA*** modificar el archivo ***hosts*** añadiendo la línea.

```
127.0.0.1 *.internetdownloadmanager.com
```

El problema es que ***IDM*** antes de validar el ***SERIAL*** a través de internet, revisa el archivo ***hosts*** y si encuentra una entrada con su dominio la comenta dejándola inválida.

```
#27.0.0.1 *.internetdownloadmanager.com
```

Todo el procedimiento de como valida nuestro serial a través de sus servidores y la

detección del dominio en el archivo hosts lo veremos en la próxima parte, por ahora me despido.