



## 1.- INTRODUCCIÓN.

Bueno para comenzar con los Niveles de [Unpacking orientado a Troyanos](#) puse éste empaquetador RE-conocido (como diría una amiga argentina), inicio este proyecto con la intención de que siga adelante y con la ilusión de que sirva para algo. Explicaré los métodos utilizados, pero lo más importante, PORQUE se hace así, que es lo que llevó a hacerlo de esa manera y por consiguiente la solución. Con ésta breve introducción os iré acercando poco a poco hacia nuestro objetivo ☺.

## 2.- CONOCIENDO EL EMPAQUETADO Y EL SERVER.

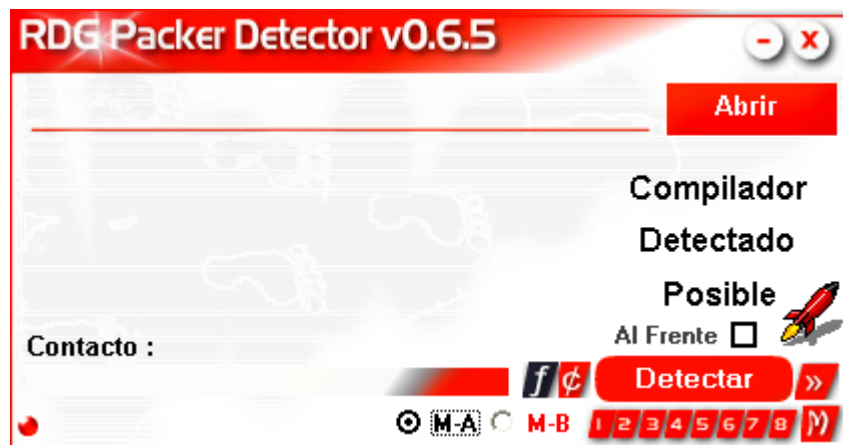
Lo primero que debemos hacer cuando tenemos un server es averiguar con que se empaquetó, existen varios detectores de archivos empaquetados, a lo largo de éste tiempo atrás se han utilizado unos pocos (con normalidad). Aunque ahora mismo el [RDG Packer Detector](#) ha eclipsado a todos, cabe destacar algunos:

- [Language 2000.](#)
- [PE Scan.](#)
- [PEiD.](#)
- [RDG Packer Detector.](#)

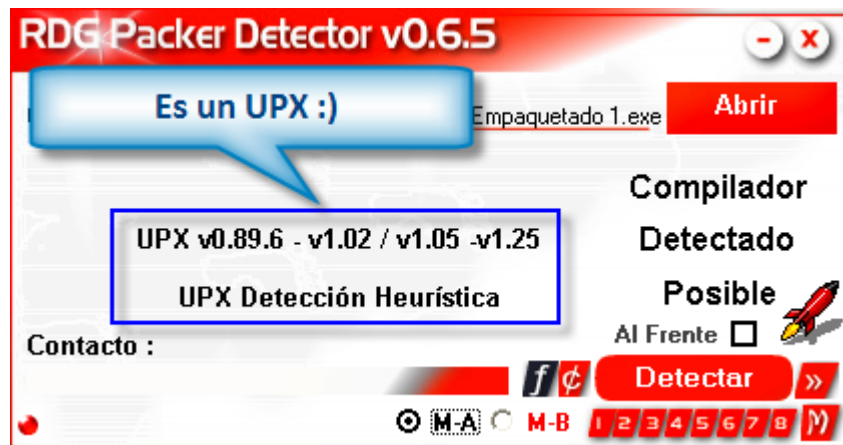
Básicamente con estos 4 se han solido detectar la mayoría de las veces, ahora mismo los 2 primeros están obsoletos, y “PEiD” sigue sin actualizarse y poco a poco va perdiendo fiabilidad, actualmente el mejor de todos ellos es sin duda el “RDG Packer Detector”. Así que de ahora en adelante únicamente utilizaremos el que acabo de mencionar.

#### [RDG PACKER DETECTOR \[ACT: 02 Abril 2007\] \(ÚLTIMA VERSIÓN\)](#)

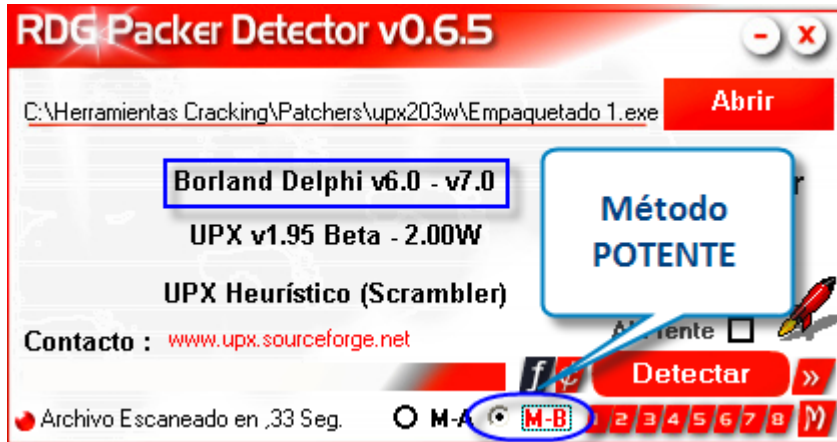
Una vez lo hayáis descargado ya podemos analizar cualquier server, cualquier programa. El método es sencillo ya que RDG soporta Drag and Drop podemos arrastrar el ejecutable hacia el mismo programa abierto.



Arrastráis el “Empaquetado 1.exe” hacia el RDG y nos mostrará el tipo de empaquetado que lleva.



Como véis analizándolo vemos que es un UPX, sin embargo no siempre acertará en el “Método rápido” (M-A) por lo tanto nos acostumbraremos a utilizar el “Método potente – Multidetección” (M-B), así que pinchamos en M-B para hacer un escaneo más ajustado.



Si os fijáis ahora nos da una información mas detallada sobre el tipo de empaquetado, incluso se ha atrevido a decirnos el tipo de compilador ([Borland Delphi v6.0 – v7.0](#)), y también nos da con exactitud la versión de nuestro UPX, ([1.95 Beta – 2.00W](#)). Ahora tenemos la información necesaria para meternos de lleno al desempaquetado.

### 3.- DESEMPAQUETANDO.

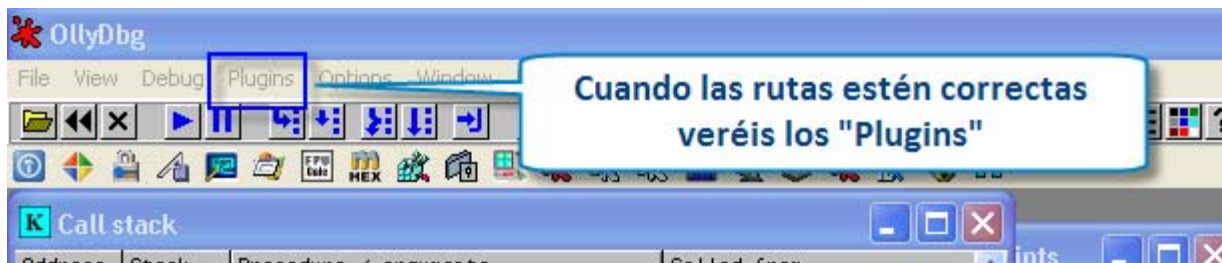
#### 3.1- TEORÍA.

El objetivo de ésta sección es que entendáis “**COMO**” se desempaqueta, y “**PORQUE**” se desempaqueta de ésta manera. Porque para hacer lo mismo hay muchos métodos diferentes, pero quiero que entendáis porque se hace así, y como se llega a descubrir.

Bien, lo que haremos será abrirlo con **OllYDBG**, y así nos iremos calentando un poco con el entorno.

#### [OllYDBG por Shaddy COMPLETO \(PLUGINS, COLORES, ETC..\)](#)

Una vez lo descarguéis tenéis que configurar la ruta de los plugins, hasta que no esté bien configurada no veréis en el menu la palabra “**Plugins**”.



Y si ya lo tenemos configurado podemos abrir el “[Empaquetado 1.exe](#)” para ver que forma tiene.

The screenshot shows the CPU window of Immunity Debugger with the following panes and callouts:

- VENTANA DE CÓDIGO / DEENSAMBLADO**: Points to the disassembly pane on the left, showing assembly instructions like `mov esi, Empaquet.0044A0D0`.
- VENTANA DE REGISTROS**: Points to the registers pane on the right, showing the state of CPU registers like `EAX 00000000` and `ECX 0012FFB0`.
- VENTANA DE INFORMACIÓN**: Points to the information pane at the bottom left, showing memory dump data in hexadecimal and ASCII.
- ESTA ES LA VENTANA DEL "DUMP" (MEMORIA)**: Points to the memory dump pane at the bottom left, showing a hex dump of memory.
- VENTANA DE LA PILA / STACK**: Points to the stack pane on the right, showing the current stack frame with `0012FFC4` as the address.

Quedaos bien con el nombre de cada ventana porque a partir de ahora cada ventana será nombrada como pone en la Imagen, así que si digo buscad **EAX** en la ventana de los registros, espero que me entendáis sin ningún problema.

Ahora es la hora de la teoría, imagino que muchos no sabéis apenas ensamblador, y claro, será difícil explicar porque hace esto el empaquetado. Sin embargo me esforzaré al máximo para que quede claro el concepto.

Cuando un programa es cargado, contiene una serie de direcciones en los **REGISTROS**, éstas direcciones son necesarias para iniciar la aplicación, ya que contienen **punteros** (direcciones que APUNTA a zonas de memoria).

```

Registers (FPU)
EAX 00000000
ECX 0012FFB0
EDX 7C91EB94 ntdll.KiFastSystemCallRet
EBX 7FFD8000
ESP 0012FFC4
EBP 0012FFFF
ESI FFFFFFFF
EDI 7C920738 ntdll.7C920738
EIP 00472CA0 Empaquet.<ModuleEntryPoint>
C 0 ES 0023 32bit 0 (FFFFFFFF)
P 1 CS 001B 32bit 0 (FFFFFFFF)
A 0 SS 0023 32bit 0 (FFFFFFFF)
Z 1 DS 0023 32bit 0 (FFFFFFFF)
S 0 FS 003B 32bit 7FDF000 (FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_FILE_NOT_FOUND (00000002)
EFL 00000246 (NO,NB,F,BE,S,PE,GE,LE)
ST0 empty -L
ST1 empty 0.
ST2 empty 0.
ST3 empty 0.
ST4 empty 0.
ST5 empty 0.
ST6 empty 1.
ST7 empty 1.
FST 4020 Cond 1 0 0 0 Err 0 0 1 0 0 0 0 (EQ)
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1 1

```

Una breve descripción de los registros.

**EAX** = Acumulador.

**ECX** = Contador.

**EDX** = Datos.

**EBX** y **EBP** = Registros de base, el de uso general es EBX.

**ESI** y **EDI** = (ESI = Source; EDI = Destiny) Registros de índice.

**ESP** y **EIP** = Registros de Puntero. (ESP a la pila y EIP a la dirección de ejecución del programa).

Aunque no se comprenda del todo la explicación de los registros, se entiende que los **3 primeros** son utilizados mas por el programa que por el Sistema en sí. Así que solamente quiero que tengáis una mera idea de cuáles son los registros que observaremos siempre como utilizados por el programa y cuáles por el sistema.

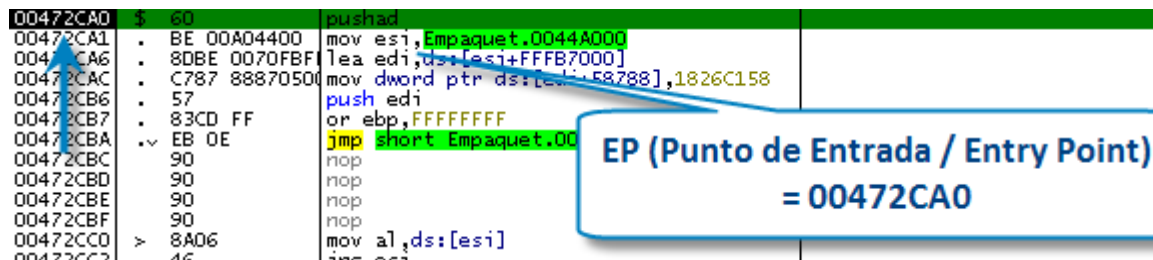
Bien, ahora imaginad, que queréis vosotros empaquetar un .exe, tenéis miles de líneas de código que queréis dejar con menor tamaño, tenéis un algoritmo que las reduce un 37%, es decir, el correspondiente método de **Compresión / Descompresión** (el utilizado por WinRAR no se aleja mucho del utilizado por un UPX), así que el proceso sería el siguiente: Una vez tenemos el .EXE comprimido, el mismo se tendría que descomprimir al arrancarse (**AUTOEJECUTABLE**).

Cuando está **sin arrancar** el código se encuentra **comprimido**, si observásemos un poco el código veríamos que se encuentra todo lleno de saltos y zonas que hacen cosas bastante incomprensibles.

Ahora quiero que imaginéis éste proceso.

- Se arranca el programa.
- Se Descomprime el código del .EXE original.
- Se deja el programa PARADO en el Punto de Entrada Original (OEP).
- Con todos los registros preparados para arrancar (es decir, todo correcto)
- El programa ya quedaría como si no tuviese compresor y por lo tanto arrancaría.

El OEP (Punto de Entrada Original / Original Entry Point) no es más que la dirección que tenía el programa antes de ser comprimido. Ahora en este caso el nuevo Entry Point (EP; Punto de Entrada) es la primera línea que nos muestra OllyDBG.



```
00472CA0 $ 60 pushad
00472CA1 . BE 00A04400 mov esi,Empaquet.0044A000
00472CA6 . 8DBE 0070FBF lea edi,ds:[esi+FFFB7000]
00472CAC . C787 8887050 mov dword ptr ds:[edi+58788],1826C158
00472CB6 . 57 push edi
00472CB7 . 83CD FF or ebp,FFFFFFFF
00472CBA . EB 0E jmp short Empaquet.00472CA0
00472CBC . 90 nop
00472CBD . 90 nop
00472CBE . 90 nop
00472CBF . 90 nop
00472CC0 > 8A06 mov al,ds:[esi]
00472CC3 . 4C inc ecx
```

Sin embargo el programa al arrancar **descomprimirá** el código y una vez lo tenga descomprimido lo enviará hacia el **Punto de entrada** que tenía **antes de ser comprimido**, así todo estaría **como al principio**. Con ésta teoría quiero que comprendáis lo siguiente.

El Compresor necesita **preservar** los registros de inicio, es decir, EAX, EBX, ECX, etc. Al arrancar tienen unos valores, ahora si el programa empieza a descomprimir código todos ellos irán **cambiando por su uso** (se irán modificando los valores). Y si llama **una vez descomprimido** al OEP (**Punto de entrada original**) y no están los valores que tenía al inicio el programa, **no arrancará** y nos dará un fallo. Así que el método CLÁSICO de desempaquetado en Packer tipo Compresor, es el llamado **PUSHAD / POPAD**.

Éste método es basado en que el compresor hace un PUSHAD.

**PUSHAD** = Mete todos los valores del registro a la pila (y así los **preservamos** 😊).

**POPAD** = Devuelve todos los valores al registro.

Es decir, el programa hace.

- **PUSHAD** (Guarda los registros)
- **CODIGO DE DESCOMPRESION**

- POPAD (Recupera los registros)
- OEP (Punto de Entrada Original)

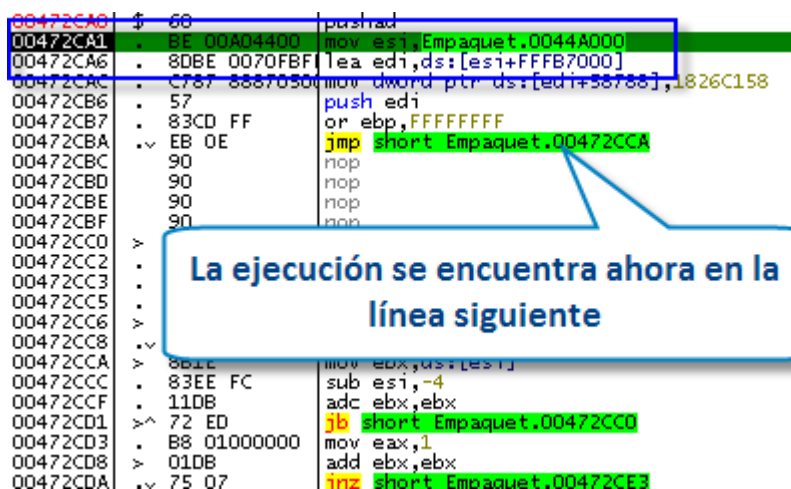
De esta manera **no compromete al programa original** y lo deja **tal y como arrancaría si no tuviese el compresor**.

### 3.2- PRACTICA.

Espero que se haya entendido la teoría. Ahora miraremos la práctica. El proceso será el siguiente:

- Pasaremos el PUSHAD con **F8** (Step Over).
- Pondremos un **Hardware Breakpoint On Access** en la dirección que puso el PUSHAD en la pila (es el puntero [dirección que apunta a] a **donde se guardaron los valores de los registros**).
- Una vez pare (en el POPAD), **tracearemos hasta** ver el **Punto de Entrada Original** (OEP).
- Volcaremos el ejecutable y repararemos la tabla de importaciones.
- Analizaremos que todo quedó correcto.

Comencemos entonces, pulsemos **F8** (NOTA: Da lo mismo en este caso dar a F8 que a F7 la diferencia entre ambos es que F7 se utiliza para entrar en los CALL y F8 los salta), y miremos donde se alojó el puntero.



```
00472CA8  $ 60      pushad
00472CA9  . BE 00A04400 mov esi,Empaquet.0044A000
00472CA6  . 8DBE 0070FBF1 lea edi,ds:[esi+FFFB7000]
00472CAE  . C787 88870500 mov dword ptr ds:[edi+58788],1826C158
00472CB6  . 57      push edi
00472CB7  . 83CD FF    or ebp,FFFFFFFF
00472CBA  . 74 0E     jmp short Empaquet.00472CCA
00472CBC  . 90      nop
00472CBD  . 90      nop
00472CBE  . 90      nop
00472CBF  . 90      nop
00472CC0  . 74 0E     jmp short Empaquet.00472CCA
00472CC2  . 74 0E     jmp short Empaquet.00472CCA
00472CC3  . 74 0E     jmp short Empaquet.00472CCA
00472CC5  . 74 0E     jmp short Empaquet.00472CCA
00472CC6  . 74 0E     jmp short Empaquet.00472CCA
00472CC8  . 74 0E     jmp short Empaquet.00472CCA
00472CCA  . 8B1C     mov ebx,ds:[esi]
00472CCC  . 83EE FC    sub esi,-4
00472CCF  . 11DB     adc ebx,ebx
00472CD1  . 72 ED     jb short Empaquet.00472CC0
00472CD3  . B8 01000000 mov eax,1
00472CD8  . 01DB     add ebx,ebx
00472CDA  . 75 07     jnz short Empaquet.00472CE3
```

Si nos fijamos en la ventana de los registros veremos que uno de ellos **se puso de color rojo**.



```

Registers (FPU)
EAX 00000000
ECX 0012FFB0
EDX 7C91EB94 ntdll.KiFastSystemCallRet
EBX 7FFDB888
ESP 0012FFA4
EBP 0012FFB0
ESI FFFFFFFF
EDI 7C920738 ntdll.7C920738
EIP 00472CA1 Empaquet
C 0 ES 0023 32bit 00
P 1 CS 001B 32bit 00
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_FILE_NOT_FOUND (00000002)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty -UNORM BA44 01050104 00000000
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 1.00000000000000000000
ST7 empty 1.00000000000000000000
FST 4020 Cond 1 0 0 0 Err 0 0 1 0 0 0 0 (EQ)
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1 1

```

ESP = PUNTERO A LA PILA

Así que ahora miraremos la ventana de [la pila](#) (Stack) para ver si realmente puso esa dirección :P.

```

0012FFA4 7C920738 ntdll.7C920738
0012FFA8 FFFFFFFF
0012FFAC 0012FFB0
0012FFB0 0012FFC4
0012FFB4 7FFD0000
0012FFB8 7C91EB94 ntdll.KiFastSystemCallRet
0012FFBC 0012FFB0
0012FFC0 00000000
0012FFC4 7C816FE0 RETURN to kernel
0012FFC8 7C920738 ntdll.7C920738
0012FFCC FFFFFFFF
0012FFD0 7FFD0000
0012FFD4 8054A6E0
0012FFD8 0012FFC8
0012FFDC 89D47B38
0012FFE0 FFFFFFFF End of SEH chain
0012FFE4 7C839AA8 SE handler
0012FFE8 7C816FE0 kernel32.7C816FE0
0012FFEC 00000000
0012FFF0 00000000
0012FFF4 00000000
0012FFF8 00472CA0 Empaquet.<ModuleEntryPoint>
0012FFFC 00000000

```

TANTO EL PUNTERO COMO LOS REGISTROS SE GUARDARON :)

Si os fijáis 12FFA4 es la dirección que contiene la pila (ESP) que es la que [apunta a los registros que teníamos antes](#), los podemos ver en las 7 líneas posteriores.

0012FFA4 7C920738 ntdll.7C920738 [EDI]

0012FFA8 FFFFFFFF [ESI]

0012FFAC 0012FFB0 [EBP]

0012FFB0 0012FFC4 [ESP]



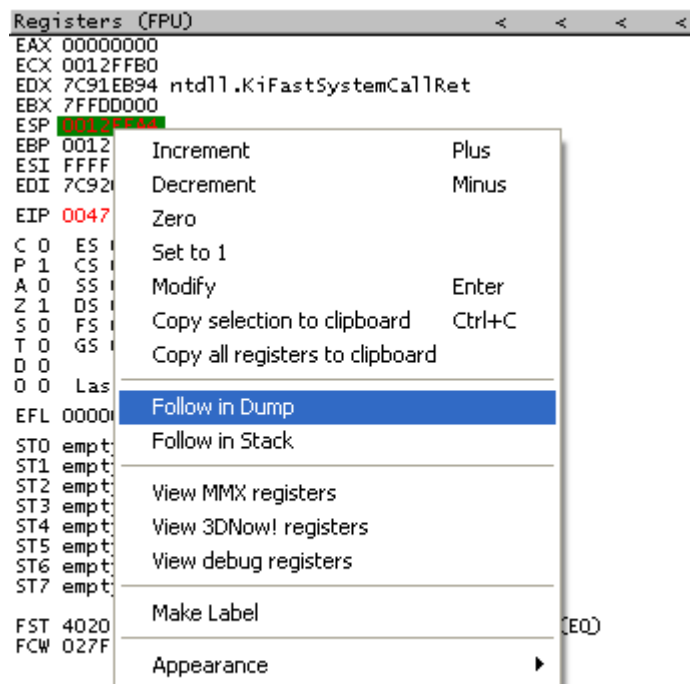
0012FFB4 7FFDD000 [EBX]

0012FFB8 7C91EB94 ntdll.KiFastSystemCallRet [EDX]

0012FFBC 0012FFB0 [ECX]

0012FFC0 00000000 [EAX]

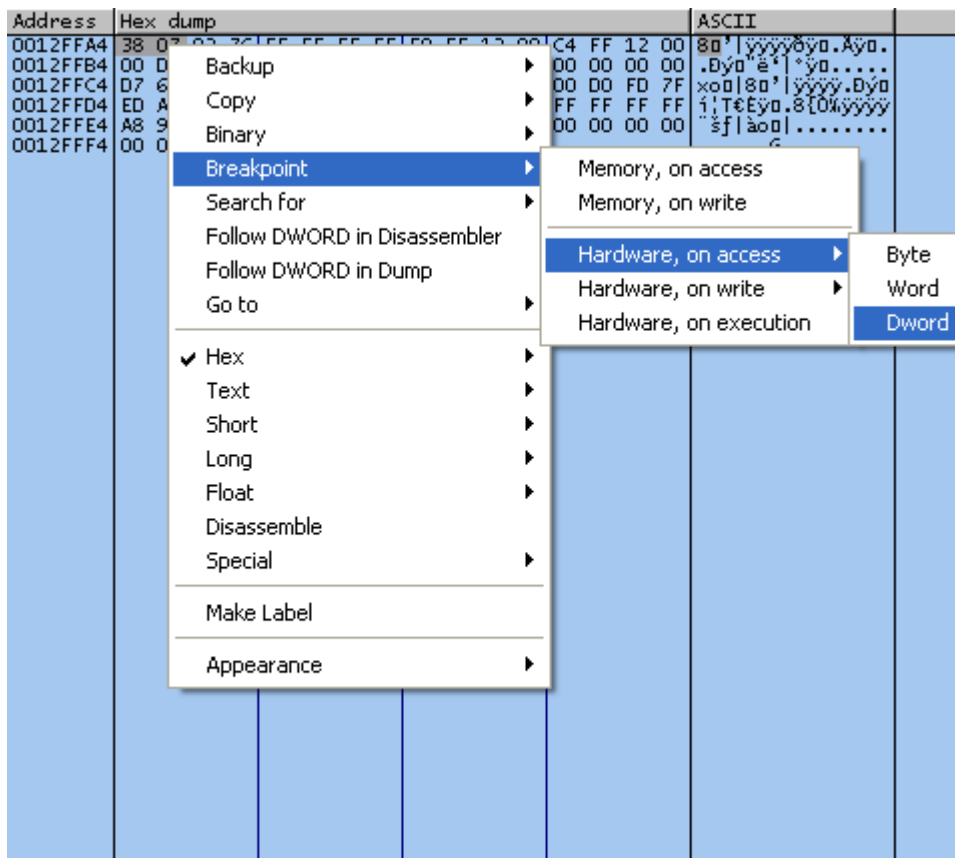
Ahora que vemos TODOS los registros en la ventana de la pila, pasaremos el de ESP (el que apunta a la pila) para ponerle el [Hardware BreakPoint On Access](#). Seleccionamos en la ventana de registros el registro ESP le damos al botón derecho y le damos a “Follow in dump” (Encontrar en la memoria).



Y ahora tendremos ya situada la ventana del [dump](#) en la dirección correspondiente de la pila.

Address	Hex dump	ASCII
0012FFA4	33 07 92 7C FF FF FF FF	8a'   yyy y y y . A y a .
0012FFB4	00 00 FD 7F 94 EB 91 7C B0	. D y a ' e '   ' y a . . . .
0012FFC4	07 6F 81 7C 38 07 92 7C FF	x o a   8 a '   y y y y . D y a
0012FFD4	ED A6 54 80 C8 FF 12 00 38	i ; T e E y a . 8 [ 0 y y y y
0012FFE4	A8 9A 83 7C E0 6F 81 7C 00	' s f   a o a   . . . . .
0012FFF4	00 00 00 00 A0 2C 47 00 00	. . . . , G . . . .

Así que seleccionamos los 2 primeros bytes (podéis seleccionar los 4 es lo mismo xD), botón derecho, [Hardware Breakpoint On Access Dword](#).



Con esto conseguiremos que el programa pare cuando devuelva los registros a como estaban al principio. Así que ahora le damos a **Play** (F9) y vemos donde nos lleva ☺.

00472E46	> 804424 80	lea eax,ss:[esp-80]
00472E4A	> 6A 00	push 0
00472E4C	. 39C4	cmp esp,eax
00472E4E	^ 75 FA	jnz short Empaquet.00472E4A
00472E50	. 83EC 80	sub esp,-80
00472E53	^ E9 A858FEFF	jmp Empaquet.00458700
00472E58	702E4700	dd Empaquet.00472E70
00472E5C	A42E4700	dd Empaquet.00472EA4
00472E60	88974500	dd Empaquet.00459788
00472E64	00	db 00
00472E65	00	db 00
00472E66	00	db 00
00472E67	00	db 00
00472E68	00	db 00
00472E69	00	db 00
00472E6A	00	db 00
00472E6B	00	db 00

Éste es un momento clave, muchos packer cuando paran después del popad, si os fijáis y subís una línea mas arriba encontraréis esto.

**- [CPU - main thread, module Empaquet]**

File View Debug Plugins Options Window Help

LEMTWHC/KBR...S

00472E3D	. 50	push eax
00472E3E	. 54	push esp
00472E3F	. 50	push eax
00472E40	. 53	push ebx
00472E41	. 57	push edi
00472E42	. FFD5	call ebp
00472E44	. 58	pop eax
00472E45	. 61	popad
00472E46	> 6A 00	push 0
00472E4A	. 39C4	cmp esp,eax
00472E4C	. 75 FA	jnz short Empaquet.00472E50
00472E50	. 83EC 80	sub esp,-80
00472E53	. E9 A858FEFF	jmp Empaquet.00458700
00472E58	. 702E4700	dd Empaquet.00472E70
00472E5C	. A42E4700	dd Empaquet.00472E4A
00472E60	. 88974500	dd Empaquet.00459788
00472E64	. 00	db 00
00472E65	. 00	db 00
00472E66	. 00	db 00
00472E67	. 00	db 00
00472E68	. 00	db 00
00472E69	. 00	db 00
00472E6A	. 00	db 00
00472E6B	. 00	db 00
00472E6C	. 00	db 00
00472E6D	. 00	db 00
00472E6E	. 00	db 00
00472E6F	. 00	db 00
00472E70	. 00	db 00
00472E71	. 00	db 00

**VUELVEN A TENER LOS MISMOS VALORES**

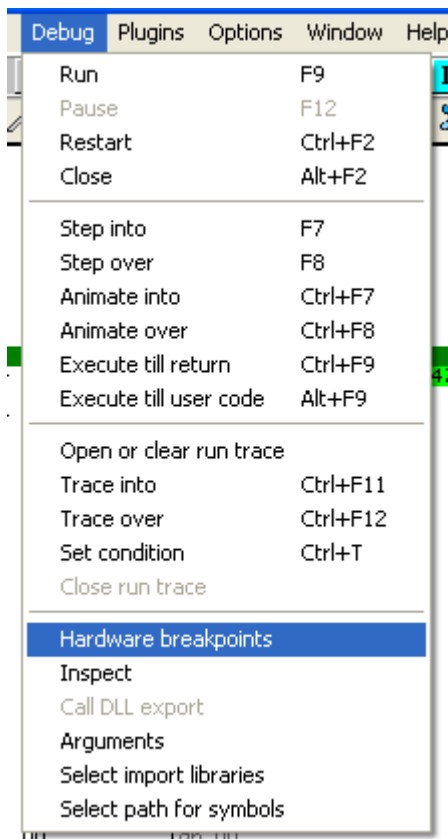
**POPAD :)**

**Registers (MMX)**

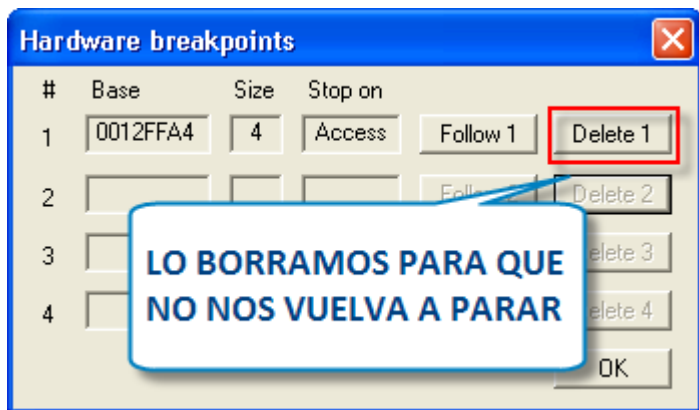
EAX	00000000
ECX	0012FFB0
EDX	7C91EB94 ntdll.KiFa
EBX	7FFD6000
ESP	0012FFC4
EBP	0012FFFF
ESI	FFFFFFFF
EDI	7C920738 ntdll.7C92
EIP	00472E46 Empaquet.0
C 0	ES 0023 32bit 0(FF
P 0	CS 001B 32bit 0(FF
A 0	SS 0023 32bit 0(FF
Z 0	DS 0023 32bit 0(FF
S 0	FS 003B 32bit 7FFD
T 0	GS 0000 NULL
D 0	
O 0	LastErr ERROR_FILE
EFL	00000202 (NO,NB,NE,
MM0	0105 0104 0000 0000
MM1	0000 0000 0000 0000
MM2	0000 0000 0000 0000
MM3	0000 0000 0000 0000
MM4	0000 0000 0000 0000
MM5	0000 0000 0000 0000
MM6	8000 0000 0000 0000
MM7	8000 0000 0000 0000

En éste momento, aunque no estéis con UPX, cualquier packer que utilice el método PUSHAD/POPAD **siempre os llevará al OEP**, de alguna manera u otra, bien sea con un RETN, un JMP EAX, o un JMP OEP, hay miles de maneras de llegar (consultar un tutorial de ensamblador y aplicar la creatividad), así que bueno, yo creo que veis la última instrucción, esta bastante claro que **ese JMP nos llevará hasta el OEP** ¿Por qué?, bien, estamos en la dirección 472E45 (Dirección perteneciente a código del packer) y nos lleva hasta 458700 casi **20000 bytes de diferencia**.

Lo primero que vamos a hacer es **quitar el Hardware Brekpoint**, para ello vamos al **Menú, Debug** y seleccionamos **Hardware Breakpoints**.



Y borramos el BP.



así que básicamente lo que haremos es poner un **Temporal BreakPoint (F4)** éste BreakPoint se caracteriza en que es como uno normal (F2) pero con la diferencia que **al parar en la zona automáticamente se desactiva** (por eso no es siquiera marcado), así que lo que haremos será llevar la ejecución hasta esa línea y desde ahí pulsaremos F7.

```

00472E44 . 58      pop eax
00472E45 . 61      popad
00472E46 . 8D4424 80 lea eax,ss:[esp-80]
00472E4A v 6A 00    push 0
00472E4C . 39C4     cmp esp,eax
00472E4E . ^ 75 FA    jnz short Empaquet.00472E4A
00472E50 . 83EC 80  sub esp,80
00472E53 . ^ E9 A858FEFF jmp Empaquet.00458700
00472E58 . 702F4700 dd Empaquet.00472E70
00472E5C . A42E4700 dd Empaquet.00472EA4
00472E60 . 88974500 dd Empaquet.00459788
00472E64 . 00       db 00
00472E65 . 00       db 00
00472E66 . 00       db 00
00472E67 . 00       db 00
00472E68 . 00       db 00
00472E69 . 00       db 00
00472E6A . 00       db 00

```

SALTO AL OEP

Ahora pulsemos **F7** para saltar al **OEP** (también sería lo mismo F8).

```

00458700 55      push ebp
00458701 8BEC     mov ebp,esp
00458703 83C4 F0  add esp,-10
00458706 B8 A06F4500 mov eax,Empaquet.00456FA0
00458708 E8 18DDFAFF call Empaquet.00406428
00458710 A1 C0A94500 mov eax,ds:[45A9C0]
00458715 8B00     mov eax,ds:[eax]
00458717 E8 309EFFFF call Empaquet.0045254C
0045871C A1 C0A94500 mov eax,ds:[45A9C0]
00458721 8B00     mov eax,ds:[eax]
00458723 BA 60874500 mov edx,Empaquet.00458760
00458728 E8 B798FFFF call Empaquet.00451FE4
0045872D 8B0D A8AA4500 mov ecx,ds:[45AA8]
00458733 A1 C0A94500 mov eax,ds:[45A9C0]
00458738 8B00     mov eax,ds:[eax]
0045873A 8B15 686D4500 mov edx,ds:[456D68]
00458740 E8 1F9EFFFF call Empaquet.00452564
00458745 A1 C0A94500 mov eax,ds:[45A9C0]
0045874A 8B00     mov eax,ds:[eax]
0045874C E8 939EFFFF call Empaquet.004525E4
00458751 E8 FEBDFAFF call Empaquet.00404554
00458756 0000     add ds:[eax],al
00458758 FFFF     ???
0045875A FFFF     ???
0045875C 0F0000   sltd ds:[eax]
0045875F 0043 75  add ds:[ebx+75],al
00458762 v 72 73    jb short Empaquet.004587D7
00458764 6F       outs dx,dword ptr es:[edi]
00458765 2055 6E  and ss:[ebp+6E],dl
00458768 v 70 61    jo short Empaquet.004587CB
0045876A 636B 69  arpl ds:[ebx+69],bp
0045876D 6E       outs dx,byte ptr es:[edi]
0045876E 67:0000  add ds:[bx+si],al
00458771 0000     add ds:[eax],al
00458773 0000     add ds:[eax],al
00458775 0000     add ds:[eax],al
00458777 0000     add ds:[eax],al
00458779 0000     add ds:[eax],al
0045877B 0000     add ds:[eax],al

```

ASCII "Curso Unpacking"  
Empaquet.0045E5D0


ORIGINAL ENTRY POINT  
(PUNTO DE ENTRADA ORIGINAL)

I/O command

I/O command

Ahora lo veréis así si no tenéis analizado el código, yo lo que suelo hacer (si el packer nos lo permite claro jeje) es analizar el código (**CTRL + A**) y tomará ésta forma.

```
00458700 > .55 push ebp
00458701 . 8BEC mov ebp,esp
00458703 . 83C4 F0 add esp,-10
00458706 . B8 A06F4500 mov eax,Empaquet.00456FA0
00458708 . E8 18DDFAFF call Empaquet.00406428
00458710 . A1 C0A94500 mov eax,ds:[45A9C0]
00458715 . 8B00 mov eax,ds:[eax]
00458717 . E8 309EFFFF call Empaquet.0045254C
0045871C . A1 C0A94500 mov eax,ds:[45A9C0]
00458721 . 8B00 mov eax,ds:[eax]
00458723 . BA 60874500 mov edx,Empaquet.00458760 ASCII "Curso Unpacking"
00458728 . E8 B798FFFF call Empaquet.00451FE4
0045872D . 8B00 A8AA4500 mov ecx,ds:[45AA8] Empaquet.0045E5D0
00458733 . A1 C0A94500 mov eax,ds:[45A9C0]
00458738 . 8B00 mov eax,ds:[eax]
0045873A . 8B15 686D4500 mov edx,ds:[456D68] Empaquet.00456DB4
00458740 . E8 1F9EFFFF call Empaquet.00452564
00458745 . A1 C0A94500 mov eax,ds:[45A9C0]
0045874A . 8B00 mov eax,ds:[eax]
0045874C . E8 939EFFFF call Empaquet.004525E4
00458751 . E8 FEBDFAFF call Empaquet.00404554
00458756 . 0000 add ds:[eax],al
00458758 . FFFFFFFF dd FFFFFFFF
0045875C . 0F000000 dd 0000000F
00458760 . 43 75 72 73 66 ascii "Curso Unpacking",0
00458770 . 00 db 00
00458771 . 00 db 00
00458772 . 00 db 00
00458773 . 00 db 00
00458774 . 00 db 00
00458775 . 00 db 00
00458776 . 00 db 00
00458777 . 00 db 00
00458778 . 00 db 00
00458779 . 00 db 00
0045877A . 00 db 00
0045877B . 00 db 00
0045877C . 00 db 00
0045877D . 00 db 00
```



Y bueno con la experiencia veréis que al ver ésta imagen reconocéis un OEP al vuelo, casi todos empiezan por un **PUSH EBP**, o **PUSH 0**, o alguna instrucción con ese tipo de estructura, son todos bastante parecidos.

Bueno por el momento ya valió, ahora tenemos la mitad hecha, encontramos el OEP, lo siguiente que haremos en la Parte II será arreglar la tabla de importaciones (ya explicaré que es y para que se utiliza) y veremos un poco como calcularla y una mera idea sobre cómo manejarla.

HASTA LA SEGUNDA PARTE ☺.