

Inline
patching
themida
2.1.6.0

March 5

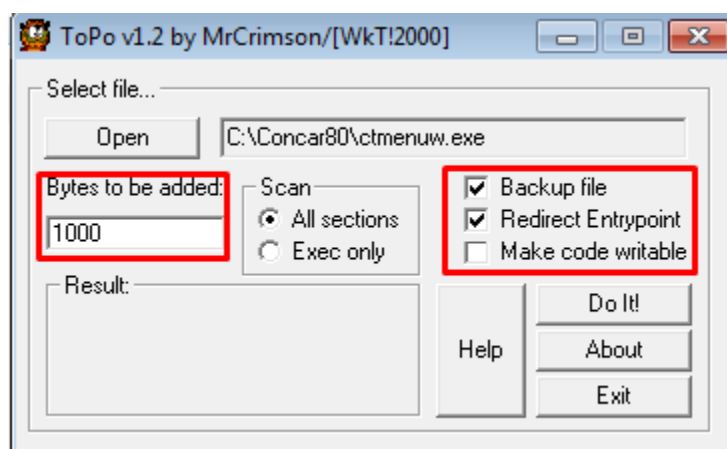
2014

torrescrack.blogspot.com

La idea de este escrito, es mostrar un método que espero e intento pueda ser “general” (sigo trabajando en ello) para aplicar inline-patching a binarios cifrados con themida, hace unas horas mande a ricardo un escrito que postee en el blog de hackingmexico, hablando sobre este mismo tema solo que aplicado en otro software y con menos explicación, le comente que escribiría uno especial para la lista mejor explicado y con mi código adjunto (al final) para hacer algunas pruebas , por si a alguien le sirve.

Hace un tiempo lvinson escribió acerca de unpack/themida de un software hecho en VB, del cual yo anteriormente había escrito solo que en esos tiempos estaba empackado con visual protect según recuerdo, probé con ese mismo software el método de inline – patching que venía trabajando y vaya que me sirvió para hacer pruebas y mejorar algunos detalles que se me habían pasado, pero no los aburro y vayamos con eso.

Bien lo primero que haremos para inlinear nuestro binario, será crear una sección de código vacía dentro del mismo, existen varias herramientas para ello, pero en este caso usaremos “TOPO”, recuerden que topo creas secciones nuevas y tiene opciones de buscar huecos y usarlos, pero nosotros crearemos una sección nueva, también tildamos algunas casillas que me facilitan el labor:



Es esencial tildar que nos guarde un backup por si hacemos algo mal y que nos rediriga el EP para que inicie desde nuestro injerto..

Les quiero comentar que el themida no tenía chequeos CRC, en caso de tenerlos existe un script que parcheo esos chequeos CRC en themida.

Ahora pausemos un rato todo y pasare a explicarles un poco mas de lo que se hará:

Como sabrán, cuando escribimos código dentro de un ejecutable este ya tiene decidido lo que cargara y las API's que usara.. El compilador es el que se encarga de crear una tabla IAT (Import Address Table), que el sistema operativo se encargara de llenar con valores que correspondan a direcciones de las API's; por lo tanto un programa ya compilado tiene su propia tabla lo que nosotros debemos hacer es intentar encontrar la dirección de kernel32.dll, con eso tendríamos al toro por los cuernos.

Comenzaremos y explicare breve y es un método algo viejo y simple; les cuento, al iniciar una aplicación el sistema operativo para cargar un proceso utiliza la API CreateProcess, por lo tanto al cargar el debugger la primer dirección en el stack o pila será de retorno a kernel32.dll, bastara en buscar la dirección donde inicia en memoria la librería y el resto solo es buscar la API GetProcAddress..

0028F8D4	76823C45	RETURN to kernel32.76823C45
0028F8D8	7FFD6000	
0028F8DC	0028F91C	
0028F8E0	774837F5	RETURN to ntdll.774837F5
0028F8E4	7FFD6000	
0028F8E8	765D9201	
0028F8EC	00000000	

Como muestro al cargar en este caso software dentro del debugger puedo notar que la primer línea del stack es una dirección de retorno a kernel32 que es este caso es 0x76823c45 en otro ordenador las direcciones pueden cambiar.

Ahora bien lo que necesitamos es encontrar el inicio de la librería kernel32, para eso es muy sencillo pues recordemos que dll y .exe tienen cabecera PE por lo tanto inician con "MZ"

Ahora si abrimos nuestro IDE RADASM y empezamos a escribir este pequeño código:

```
.code

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

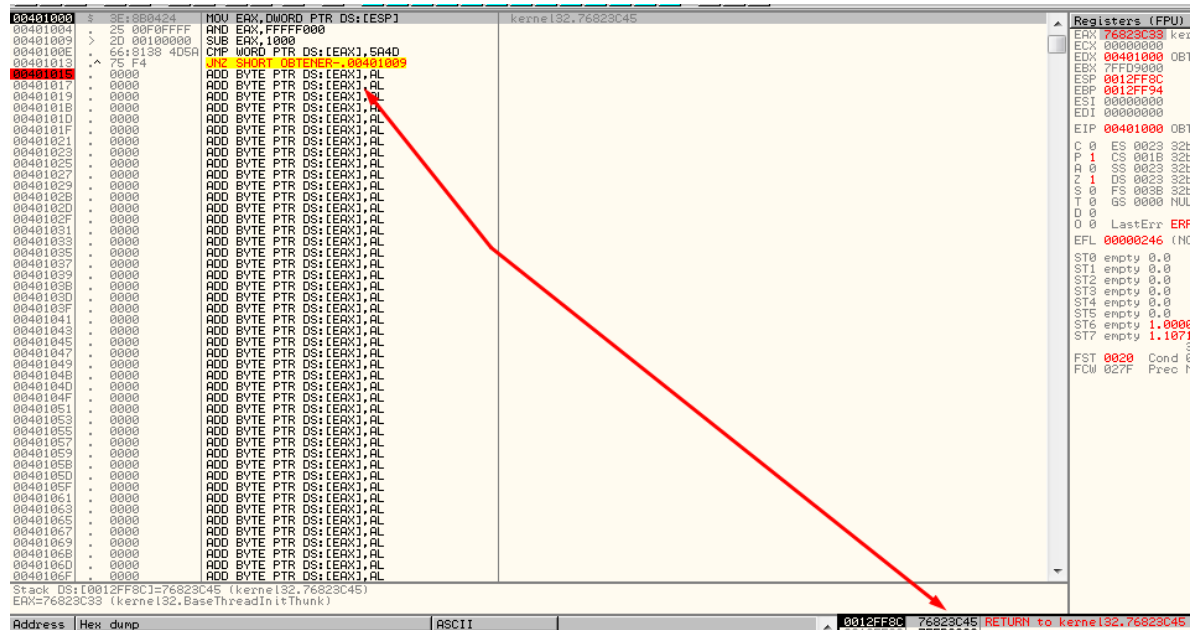
codigo:

pushad
xor ecx,ecx ;limpiamos el registro ecx
mov eax, dword ptr ds:[esp+20h] ; muevo a "EAX" la direccion de retorno al kernel32

and eax, 0FFFFFF00h ; Esto lo hacemos para descartar los últimos bytes

buscando:
sub     eax, 1000h ; vamos restando ya que como todo programa tiene que estar alineado en memoria
cmp word ptr [eax], 'ZM' ; busca los caracteres "MZ" que es el inicio de la cabecera
jnz     buscando
```

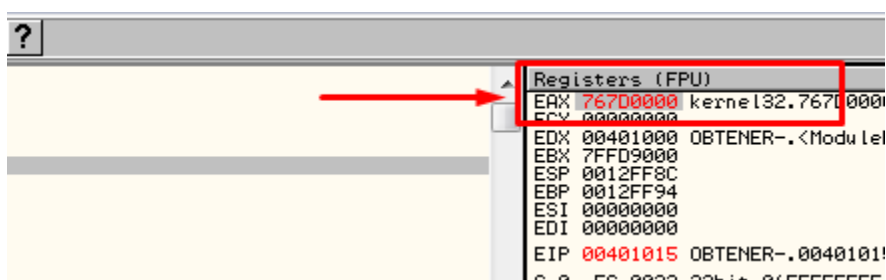
El código esta comentado, pero bueno primero guardamos todo a la pila para después recuperarlo al terminar el meollo, limpiamos el registro ecx y como se desacomoda la pila por el pushad es por eso que movemos esp +20h, después lo que se está haciendo es buscar el inicio de la librería y por ello se descartan los últimos bytes con "and eax, 0ffffff00h" y se va restando en 1000h ya que como todo programa debe estar alineado ósea que deben iniciar en una dirección múltiplo de mil, ahora bien podemos probar con olly carguemos nuestro compilado en olly y veamos:



Ahora presionamos el botón "M" (memory map) dentro de nuestro debugger y podremos verificar que kernel32 se cargó en la dirección 767D000:

767D0000	00001000	kernel32	PE header	Image R	RWE
767D1000	000C5000	kernel32	.text SFX,code,im	Image R	RWE
76896000	00001000	kernel32	.data	Image R	RWE
76897000	00001000	kernel32	.rsrc	Image R	RWE
76898000	0000C000	kernel32	.reloc	Image R	RWE
768B0000	00001000	SHLWAPI	PE header	Image R	RWE
768B1000	00051000	SHLWAPI	.text SFX,code,im	Image R	RWE
76902000	00001000	SHLWAPI	.data	Image R	RWE
76903000	00001000	SHLWAPI	.rsrc	Image R	RWE
76904000	00003000	SHLWAPI	.reloc	Image R	RWE
769FA000	00001000	MSCVF	PE header	Image R	RWE

Podemos observar que coloquemos un BreakPoint al terminar el loop para poder detener el control cuando pare su ejecución al terminar la búsqueda corremos el debugger con F9 y veamos el resultado en "EAX" y veamos si concuerda con la dirección de kernel32:



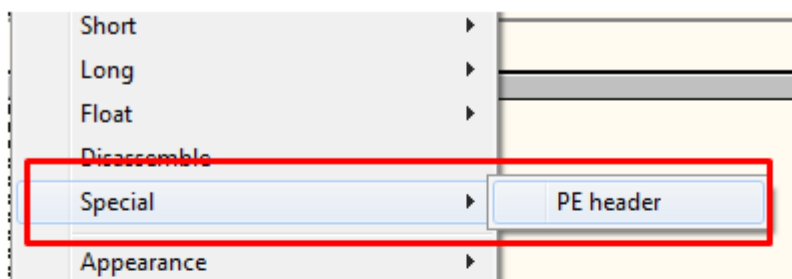
Ahora presionamos el botón "M" (memory map) dentro de nuestro debugger y podremos verificar que kernel32 se cargó en la dirección 767D0000:

767D0000	00001000	kernel32	PE header	Image R	RWE
767D1000	000C5000	kernel32	.text SFX,code,im	Image R	RWE
76896000	00001000	kernel32	.data	Image R	RWE
76897000	00001000	kernel32	.rsrc	Image R	RWE
76898000	0000C000	kernel32	.reloc	Image R	RWE
768B0000	00001000	SHLWAPI	PE header	Image R	RWE
768B1000	00051000	SHLWAPI	.text SFX,code,im	Image R	RWE
76902000	00001000	SHLWAPI	.data	Image R	RWE
76903000	00001000	SHLWAPI	.rsrc	Image R	RWE
76904000	00003000	SHLWAPI	.reloc	Image R	RWE
769FA000	00001000	MSCVF	PE header	Image R	RWE

Si vamos a esa dirección de memoria para ver el contenido podemos observar lo que estamos buscando:

Address	Hex dump	ASCII
767D0000	40 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZÉ.....
767D0010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	@.....@.....
767D0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
767D0030	00 00 00 00 00 00 00 00 00 00 00 00 F0 00 00 00F0.....
767D0040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68Th
767D0050	69 73 20 70 72 6F 67 72 61 60 20 63 61 6E 6E 6F	is program cannot
767D0060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	be run in DOS
767D0070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode.....
767D0080	63 8A 9F 9F 27 EB F1 CC 27 EB F1 CC 27 EB F1 CC
767D0090	2E 93 62 CC 16 EB F1 CC 27 EB F0 CC 55 E8 F1 CC
767D00A0	2E 93 63 CC 26 EB F1 CC 2E 93 64 CC 20 EB F1 CC
767D00B0	2F 93 72 CC 01 EB F1 CC 2F 93 75 CC 04 EB F1 CC

Ya que estamos ubicados al inicio de la librería, debemos saber que sabemos que 03Ch lugares mas adelante encontraremos la dirección (RVA) de la cabecera PE. Cambiamos la vista haciendo click derecho y buscamos 03ch lugares mas adelante:



767D003H	00	DB 00	
767D003B	00	DB 00	
767D003C	F0000000	DD 000000F0	Offset to PE signature
767D0040	0E	DB 0E	
767D0041	1F	DB 1F	
767D0042	00	DB 00	

Address	Hex dump	Data	Comment
767D00EF	00	DB 00	
767D00F0	50 45 00 00	ASCII "PE"	PE signature (PE)
767D00F4	4C01	DM 014C	Machine = IMAGE_FILE_MACHINE_I386
767D00F6	0400	DM 0004	NumberOfSections = 4
767D00F8	EFB8E74C	DD 4CE7B8EF	TimeDateStamp = 4CE7B8EF
767D00FC	00000000	DD 00000000	PointerToSymbolTable = 0
767D0100	00000000	DD 00000000	NumberOfSymbols = 0
767D0104	E000	DM 00E0	SizeOfOptionalHeader = E0 (224.)
767D0106	0221	DM 2102	Characteristics = DLL EXECUTABLE IMAGE 32BIT_MACHINE
767D0108	0B01	DM 010B	MagickNumber = PE32
767D010A	09	DB 09	MajorLinkerVersion = 9
767D010B	00	DB 00	MinorLinkerVersion = 0
767D010C	00480C00	DD 000C4800	SizeOfCode = C4800 (804864.)
767D010E	00000000	DD 00000000	SizeOfData = 00000000

Como podemos observar 03ch lugares mas adelante, vemos nos indica que el offset de la cabecera es + F0 lugares mas adelante del inicio de la libreria por lo tanto vamos a esa dirección y vemos el inicio de la cabecera, ahora debemos meter esto en el código conforme vayamos avanzando:

```
;hasta aqui tenemos la direccion de la libreria ahora vamos por busca la API
mov ebx,eax ; guarda el inicio de la libreria en el registro ebx
mov edi,eax ; guarda el inicio de la libreria en el registro edi
add ecx,03ch ; coloca en el registro ecx el valor 03ch para posteriormente sumarlo
add ebx,ecx
mov ebx, [ebx]
add eax,ebx ; sumamos el contenido para poder llegar a la cabecera
```

En los comentarios del código explico todo, que básicamente es lo que hicimos a mano pero plasmado en el código

lo que nos interesa es la tabla de exportación lo cual está ubicada siempre a 78h a partir del “PE” podríamos ver:

\$+74	10000000	DD 00000010	NumberOfRvaAndSizes = 10 (16.)
\$+78	C44F0B00	DD 000B4FC4	Export Table address = B4FC4
\$+7C	F8A70000	DD 0000A7FA	Export Table size = A7FA (43002.)
\$+80	C0F70B00	DD 000BF7C0	Import Table address = BF7C0
\$+84	F4010000	DD 000001F4	Import Table size = 1F4 (500.)

0000	Dword	Characteristics	Set to zero (currently none defined)
0004	Dword	TimeDateStamp	Often set to zero
0008	Word	MajorVersion	User-defined version number
000A	Word	MinorVersion	As above
000C	Dword	Name	RVA of DLL name (null-terminated ASCII)
0010	Dword	Base	First valid exported ordinal
0014	Dword	NumberOfFunctions	Number of entries in EAT
0018	Dword	NumberOfNames	Number of entries in ENT

001C	Dword	AddressOfFunctions	RVA of EAT (export address table)
0020	Dword	AddressOfNames	RVA of ENT (export name table)
0024	Dword	AddressOfNameOrdinals	RVA of EOT (export ordinal table)

Bien, ya estamos en la tabla de exportaciones de kernel32.dll, ahora veamos cómo está compuesta:

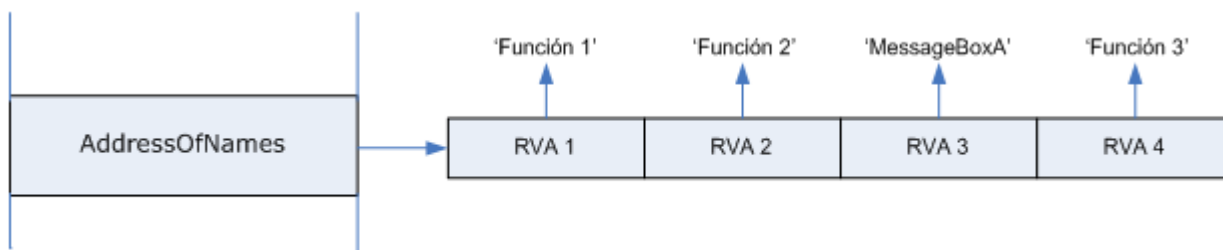
```
.....
.....
```

(Trozo tomado de un tutorial de zero)

De esta estructura interesan las direcciones de las últimas tres tablas:

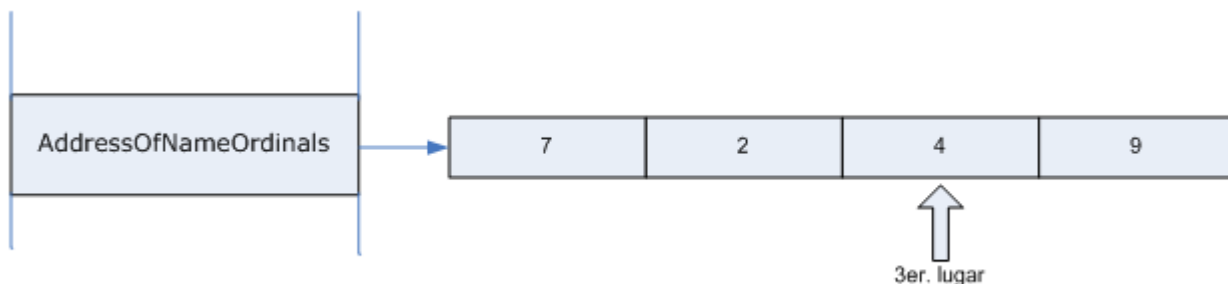
- *AddressOfFunctions*: contiene la dirección (RVA) de un listado de punteros (RVA's) a las funciones exportadas por esta librería
- *AddressOfNames*: contiene la dirección (RVA) de un listado de punteros dirigidos a los nombres de las funciones exportadas
- *AddressOfNameOrdinals*: RVA que apunta a un listado que contiene los ordinales de los nombres de las funciones exportadas

Tenemos que empezar por la tabla *AddressOfNames*, la cual contiene un listado de punteros a direcciones de memoria (RVA's) donde están guardados los nombres de las funciones exportadas:

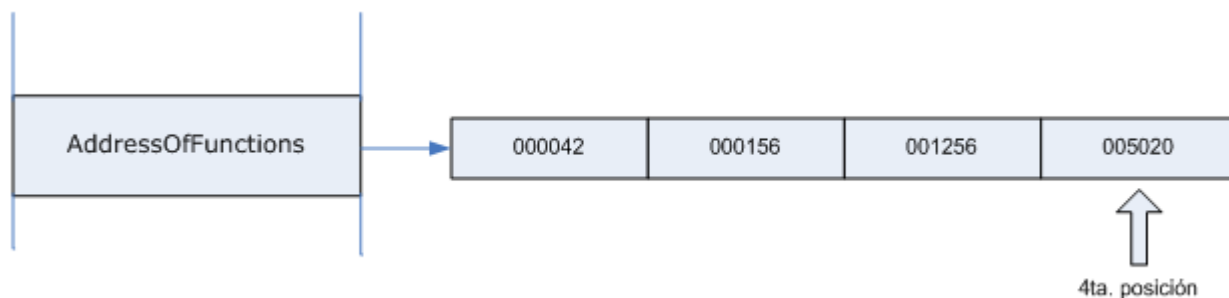


Dos acotaciones, *AddressOfNames* contiene una RVA que apunta a un listado de RVA's, no directamente a los nombres. Además recordemos que cada una de estas RVA tiene un tamaño double word.

Bien, vemos que la **tercer RVA** es la que apunta a la función buscada, ese valor nos va a servir de puntero para la próxima tabla, la *AddressOfNameOrdinals*:



En el **tercer lugar** de esta tabla tenemos un valor (en este caso hipotético es 4) que nos servirá de puntero a la última tabla, la *AddressOfFunctions*:



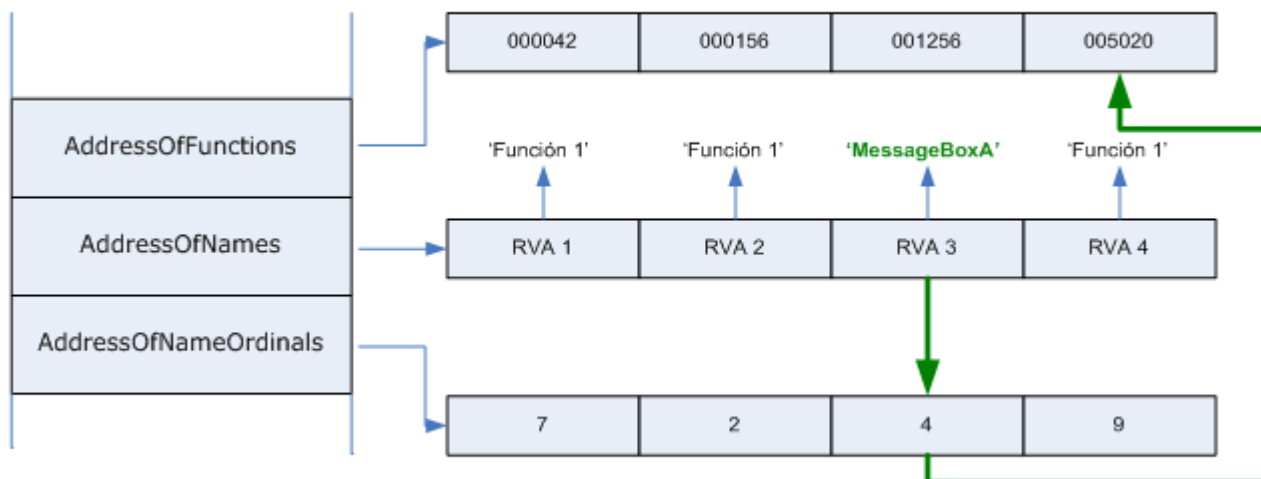
En el caso de esta tabla, los valores son del tipo *Word*, esto también lo vamos a necesitar luego cuando realicemos nuestro código.

Vemos que en la cuarta posición tenemos el valor 005020, esto nos indica la RVA donde vamos a encontrar el inicio de la función buscada en memoria, para que podamos hacer algo como esto:

```
call 405020
```

Recordemos que es una RVA, por lo que tendremos que sumarle la dirección base antes de utilizarla.

Para que quede un poco mas claro, veamos todo esto en un solo paso:



Es un esquema un poco complejo, pero una vez que lo analicemos y luego cuando veamos el código va a quedar mas claro.

.....

Primero:

```
; edi se queda con el valor de inicio de la libreria

xor ebx,ebx
mov ecx, 078h
add eax,ecx ; sumamos desde la direccion el PE + 78h para encontrar la ExportTable
mov ebx, [eax] ; sacamos el RVA y lo colocamos en ebx
add ebx, edi ; sumamos la imagebase de la libreria con el RVA del ExportTable
mov eax, [ebx+20h] ; segun la tabla que mostre sacamos el RVA de AddressOfNames
add eax, edi ; aqui sumamos y obtenemos el offset de AddressOfNames
push ebx; pone en la pila el inicio de la exportable que despues usaremos
push edi ; se pone en la pila el contenido de EDI que posteriormente usaremos (inicio de libreria)

; eax contiene el offset de AddressOfNames
```

Como vemos el código está muy comentado, básicamente lo que se hace es sumar la dirección del PE + 78 que es donde se encuentra la ExportTable, obtenemos el RVA y lo sumamos con 20h que según la descripción con la tabla de arriba es la dirección de AddressOfNames, y

guardamos datos en la pila para posteriormente usarlos, lo hacemos de esta manera pues quiero que se adapte para solo copiar y pegar.. dentro del debugger (ya lo verán mas adelante)

Ahora vamos a ir recorriendo esa tabla y vamos a buscar la API 'GetProcAddress' por su nombre, para eso realizamos la siguiente rutina:

```
; eax contiene el offset de AddressOfNames
xor ebx,ebx ; preparamos nuestro contador
sub ebx,4

xor ebp,ebp ; ponemos a cero el segundo contador

BuscadorGPA:

inc ebp ; un tipo de contador (guia) que utilizaremos al entrar en las otras tablas..
add ebx,4
mov edx, [eax+ebx] ; ira toma el contenido de eax + el contador, recuerden que en eax esta el offset de AddressOfNames
pop edi ; se toma de la pila lo que edi contenia (Offset kernel32)
add edx, edi ;
push edi
mov ecx,0Eh
mov esi,edx ; mueve a ESI el puntero del primero nombre de la lista de API's
lea edi, offset GPA ; mueve a EDI el offset donde se encuentra la API que buscaremos
cld
repe cmpsb ; realiza las comprobaciones byte a byte hasta encontrar el nombre GetProcAddress
jnz BuscadorGPA
```

Preparamos dos contadores poniéndolos a cero, lo primero que hará para ir sumando cada 4 bytes y buscar el nombre de cada api, el segundo (EBP) es para tener un índice para posteriormente usarla en la siguientes tablas; lo que se hace en esta parte del código que igual esta muy comentado para que no se queden con dudas, pero les explico, básicamente lo que se hace es comparar cada 4 bytes dentro de AddressOfNames el nombre de la API "GetProcAddress" lo cual lo hace por medio de la instrucción *cmpsb*. Esta instrucción sirve para comparar cadenas de un byte, donde la primer cadena la tenemos que tener referenciada con el registro esi y la segunda con el registro edi. (se muestra en el código como se colocan en edi y esi lo correspondiente)

Por ejemplo si queremos comparar una cadena 'C' con otra 'D', tenemos que tener lo siguiente:



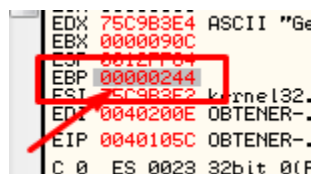
La vamos a utilizar para comparar byte a byte lo que tenemos en memoria con una variable que definimos con el nombre de la API buscada, de la forma:

```
GPA      db "GetProcAddress",0
```

Como solo compara un byte la tenemos que ponerla dentro de otra instrucción: rep (o repe), la cual repite la instrucción que le sigue la cantidad de veces que le indiquemos en el registro ecx. Por eso debemos inicializar a ecx con el largo de la cadena a comparar, en este caso GetProcAddress tiene 14 bytes, o sea que inicializamos a ecx con el valor 0Eh (14 en hexadecimal).

De toda la rutina anterior lo único que nos va a importar es justamente el contador que esta ubicado en el registro EBP, el cual vamos a utilizar como índice para la próxima tabla.

Si seguimos esta rutina dentro de ollydebugger al finalizar la rutina podríamos observar que el registro EBP (nuestro índice) guardo lo que queremos:



Ya casi acabamos. Lo que haremos ahora es obtener la dirección de la próxima tabla (AddressOfNameOrdinals), que está a 024h a partir del inicio de la tabla de exportaciones:

```

pop edi
pop ebx

dec ebp ; acomodo el contador (guia)

; ebx contiene el inicio del exportable
; edi contiene el inicio del kernel32

; hacemos lo mismo que anteriormente solo que ahora buscando la direccion de AddressOfNameOrdinals

xor ecx,ecx
mov eax, [ebx+24h]
add eax,edi

mov ecx,ebp
imul ecx,2 ; multiplico por 2

add eax,ecx

; eax esta valiando el valor sacado de AddressOfNameOrdinals

```

Recuperamos con POP edi,ebx los datos guardados anteriormente en la PILA, sacamos el RVA de AddressOfNamesOrdinals y lo sumamos con EDI que contiene el inicio del kernel32 y así obtenemos el offset de AddressOfNameOrdinals.

Ahora prepararemos el índice para acceder al valor de la tabla, para eso tenemos que multiplicar el contador EBP por dos, esto es porque cada dirección de esta tabla ocupa 2 bytes, o sea que por ejemplo el cuarto valor de esta tabla lo vamos a encontrar en la posición $4 * 2$.

Bien, ahora vamos con el final:

```

; hacemos lo mismo que anteriormente solo que ahora buscando la direccion de AddressOfFunctions

mov edx, [ebx+01ch]
add edx,edi
xor esi,esi
movzx esi, word ptr [eax]
rol esi,2 ; multiplico por 4
add edx,esi ; normalizo el RVA

mov eax,[edx]
add eax,edi

; eax tiene la direccion de la API GetProcAddress "};)"

```

Es casi lo mismo que las anteriores rutinas, le sumamos a 01ch que es donde está el RVA de AddressOfFunctions y le sumamos el inicio del kernel32 que está en EDI y luego muevo el valor del puntero que obtuvimos anteriormente (eax) a esi y lo multiplico por cuatro. Esto es similar a lo que vimos anteriormente, pero como ahora la tabla es de valores doubleword, debo multiplicar por cuatro.

En edx tenía la dirección obtenida, como es un puntero a otra dirección, lo que hago es obtener el contenido de esa dirección y luego 'normalizarla'. Al finalizar eax contendrá la dirección de la API que tanto buscábamos 😊

The screenshot shows a debugger window with assembly code on the left and a 'Registers (MMX)' window on the right. The assembly code is for a function named 'OBTENER'. The registers window shows the following values:

Register	Value
EAX	75C33303
ECX	00000486
EDX	75C958FC
EBX	75C94FC4
ESP	0012FF8C
EBP	00000243
ESI	00000910
EDI	75BE0000
EIP	00401080

Algunos se preguntaran porque me complique tanto la vida si pudo ser mas corto el codigo, pues es que la idea era que tuviéramos un codigo de tal forma que solo haciendo cambios minimos podríamos utilizarlo en cualquier otro inline a themida.

los únicos cambios que deberemos hacer al copiar y pegar el código es cambiar la dirección donde se encuentra el string "GetProcAddress" como muestro adelante:

	OBTENER-DLL.Asm	03/03/2014 05:39 ...	Easy Code Masm ...	4 KB
	OBTENER-DLL.exe	03/03/2014 05:39 ...	Aplicación	3 KB
	OBTENER-DLL.inc	08/02/2014 05:40 ...	Archivo INC	0 KB
	OBTENER-DLL.obj	03/03/2014 05:39 ...	Object File	1 KB

The screenshot shows a debugger window with assembly code on the left and a context menu on the right. The assembly code is for a function named 'OBTENER'. The context menu has the following options:

- Backup
- Copy
- Binary
- Assemble
- Label
- Comment
- Breakpoint
- Hit trace
- Run trace
- New origin here

The 'Copy' option is highlighted, and a red arrow points to the 'Binary copy' option.

Copiamos el código, ahora vamos a abrir el software en otro ollydebugger y pegamos el código en la zona creada anteriormente:

01144000	60	PUSHAD	lo que hacemos es salvar los registros para recuperarlos despues de nuestro patch
01144001	3E 8B4424 20	MOV ECX, DWORD PTR DS:[ECBP+0x20]	
01144006	25 00F0FFFF	AND EAX, 0xFFFFF000	
01144008	20 00100000	SUB EAX, 0x1000	
01144010	66 8130 4D5A	CMPL WORD PTR DS:[EAX], 0x5A4D	
01144015	75 F4	JNZ SHORT 0114400B	wvs.0114400B
01144017	8B08	MOV EBX, EAX	kernel32.BaseThreadInitThunk
01144019	8BF8	MOV EDI, EAX	kernel32.BaseThreadInitThunk
0114401B	83C1 3C	ADD ECX, 0x3C	
0114401E	03D9	ADD EBX, ECX	
01144020	8B18	MOV EBX, DWORD PTR DS:[EBX]	
01144022	03C3	ADD EAX, EBX	kernel32.BaseThreadInitThunk
01144024	8BF0	MOV ESI, EAX	
01144026	33DB	XOR EBX, EBX	
01144028	B9 70000000	MOV ECX, 0x70	
0114402D	03C1	ADD EAX, ECX	
0114402F	8B18	MOV EBX, DWORD PTR DS:[EAX]	
01144031	03DF	ADD EBX, EDI	
01144033	8B43 20	MOV EAX, DWORD PTR DS:[EBX+0x20]	
01144036	03C7	ADD EAX, EDI	
01144038	53	PUSH EBX	
01144039	57	PUSH EDI	
0114403A	33DB	XOR EBX, EBX	
0114403C	83EB 04	SUB EBX, 0x4	
0114403F	33ED	XOR EBP, EBP	
01144041	45	INC EBP	
01144042	83C3 04	ADD EBX, 0x4	
01144045	8B1403	MOV EDX, DWORD PTR DS:[EBX+EAX]	kernel32.75F53C45
01144048	5F	POP EDI	
01144049	03D7	ADD EDX, EDI	
0114404B	57	PUSH EDI	
0114404C	B9 0E000000	MOV ECX, 0xE	
01144051	8BF2	MOV ESI, EBX	wvs.(ModuleEntryPoint)
01144053	803D B8471401	LEA EDI, DWORD PTR DS:[0x11447B8]	aqui cambiamos el buffer por la direccion donde colocamos el string
01144059	FC	CALL	
0114405A	F3 A6	REPE CMPS BYTE PTR ES:[EDI], BYTE PTR DS	
0114405C	75 E3	JNZ SHORT 01144041	wvs.01144041
0114405E	5F	POP EDI	kernel32.75F53C45
0114405F	5B	POP EBX	kernel32.75F53C45
01144060	4D	DEC EBP	
01144061	8BC9	XOR ECX, ECX	
01144063	8B43 24	MOV EAX, DWORD PTR DS:[EBX+0x24]	
Address=011447B8, (ASCII "GetProcAddress")			
EDI=00000000			
Address	Hex dump	ASCII	
011447B8	47 65 74 50 72 6F 63 41	GetProcAddress	
011447C0	64 64 72 65 73 73 00 00	ddress..	
011447C8	90 90 90 E9 30 E8 FF FF	EEEE00p	
011447D0	00 00 00 00 00 00 00 00	
011447D8	00 00 00 00 00 00 00 00	
011447E0	00 00 00 00 00 00 00 00	
011447E8	00 00 00 00 00 00 00 00	
011447F0	00 00 00 00 00 00 00 00	

Bajamos y casi al final del código injertado podremos ver el jmp que nos agregó topo para redirigir a la zona donde estaba el punto de entrada y como no usaremos esa parte metemos el string "GetProcAddress" en esa dirección y modificamos muestra la imagen de arriba.

011447B3	90	NOP	
011447B4	90	NOP	
011447B5	90	NOP	
011447B6	90	NOP	
011447B7	90	NOP	
011447B8	47	INC EDI	
011447B9	65:74 50	JE SHORT 0114480C	Superfluous prefix
011447BC	72 6F	JB SHORT 0114482D	wvs.0114482D
011447BE	6341 64	ARPL WORD PTR DS:[ECX+0x64], AX	
011447C1	64:72 65	JB SHORT 01144829	Superfluous prefix
011447C4	73 73	JNB SHORT 01144839	wvs.01144839
011447C6	0000	ADD BYTE PTR DS:[EAX], AL	
011447C8	90	NOP	
011447C9	90	NOP	
011447CA	90	NOP	
011447CB	E9 30E8FFFF	JMP 01143000	wvs.01143000
011447D0	0000	ADD BYTE PTR DS:[EAX], AL	
EDI=00000000			
Address	Hex dump	ASCII	
011447B8	47 65 74 50 72 6F 63 41	GetProcAddress	
011447C0	64 64 72 65 73 73 00 00	ddress..	
011447C8	90 90 90 E9 30 E8 FF FF	EEEE00p	
011447D0	00 00 00 00 00 00 00 00	
011447D8	00 00 00 00 00 00 00 00	
011447E0	00 00 00 00 00 00 00 00	
011447E8	00 00 00 00 00 00 00 00	
011447F0	00 00 00 00 00 00 00 00	

Bien ahora guardamos los cambios y listo empezamos con lo demás que es la parte interesante, como sabrán este código lo que hace es que al terminar nos devuelva el offset de “GetProcAddress” y lo que hace esa API es devolvernos la dirección de cualquier API que le pidamos, con eso podemos sacar mas api’s, que en este caso la única que quiero es “CreateThread” que básicamente nos sirve para crear un hilo y se preguntaran ¿que haremos con ese hilo?, sencillo un hilo que vaya buscando los bytes ya descifrados en memoria (pues recordemos está cifrado) conforme sigue la ejecución del software y ya una vez encontrados procedemos a parchear, algo similar lo explico en el código de un loader simple, que mostré en el “elladodelmal”, según recuerdo;

<http://www.elladodelmal.com/2014/02/loader-simple-acceder-al-binario.html>

veamos la información de los parámetros que necesitamos meter para usar “GetProcAddress”:

```
C++
FARPROC WINAPI GetProcAddress(
    _In_ HMODULE hModule,
    _In_ LPCSTR lpProcName
);
```

Parameters

hModule [in]

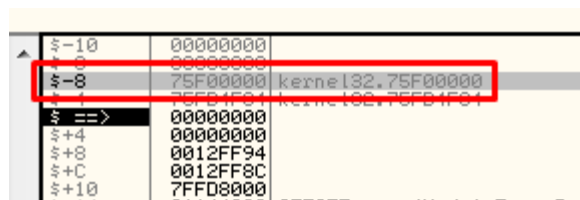
A handle to the DLL module that contains the function or variable. The [LoadLibrary](#), [LoadLibraryEx](#), [LoadPackagedLibrary](#), or [GetModuleHandle](#) function returns this handle.

The **GetProcAddress** function does not retrieve addresses from modules that were loaded using the **LOAD_LIBRARY_AS_DATAFILE** flag. For more information, see [LoadLibraryEx](#).

lpProcName [in]

The function or variable name, or the function's ordinal value. If this parameter is an ordinal value, it must be in the low-order word; the high-order word must be zero.

Solo son dos parámetros, el primero es la dirección de la dll que contiene a la API que en este caso es kernel32.dll lo cual esa dirección ya la tengo por algún lado del stack guardada, el segundo parámetro es el nombre de la API que esa la ubicaremos cerca de donde ubicamos el nombre de “GetProcAddress” veamos cómo quedaría:



0114407C	03D6	ADD EDX,ESI	
0114407E	8B02	MOV EAX,DWORD PTR DS:[EDX]	
01144080	03C7	ADD EAX,EDI	kernel32.75F00000
01144082	8B4C24 F8	MOV ECX,DWORD PTR SS:[ESP-0x8]	kernel32.75F00000
01144086	68 90471401	PUSH 0x1144790	ASCII "CreateThread"
01144088	51	PUSH ECX	kernel32.75F00000
0114408C	FFD0	CALL EAX	kernel32.GetProcAddress
0114408E	90	NOP	
0114408F	90	NOP	
01144090	90	NOP	
01144091	90	NOP	

Aquí movemos al registro ECX la dirección del kernel32 que como vemos en la pila quedo ubicado en ESP-8, después metemos los parámetro que ya había comentado y hacemos una llamada a EAX que contiene la dirección de GetProcAddress, si pasamos esas instrucciones podemos observar que el resultado está en el registro EAX devolviéndonos la dirección de la API que necesitamos:

0114404B	57	PUSH EDI	kernel32.75F00000
0114404C	B9 0E000000	MOV ECX,0xE	
01144051	8BF2	MOV ESI,EDX	kernel32.75F00000
01144053	8D3D B8471401	LEA EDI,DWORD PTR DS:[0x11447B8]	aquí cambiamos el buffer por la direccion donde colo
01144059	FC	CLD	
0114405A	F3:A6	REPE CMPS BYTE PTR ES:[EDI],BYTE PTR D	
0114405C	75 E3	JNZ SHORT 01144041	wvs.01144041
0114405E	5F	POP EDI	kernel32.75F00000
0114405F	5B	POP EBX	kernel32.75FB4FC4
01144060	4D	DEC EBP	
01144061	33C9	XOR ECX,ECX	kernel32.75F00000
01144063	8B43 24	MOV EAX,DWORD PTR DS:[EBX+0x24]	
01144066	03C7	ADD EAX,EDI	kernel32.75F00000
01144068	8BCD	MOV ECX,EBP	
0114406A	6BC9 02	IMUL ECX,ECX,0x2	kernel32.75F00000
0114406D	03C1	ADD EAX,ECX	kernel32.75F00000
0114406F	8B53 1C	MOV EDX,DWORD PTR DS:[EBX+0x1C]	
01144072	03D7	ADD EDX,EDI	kernel32.75F00000
01144074	33F6	XOR ESI,ESI	
01144076	0FB730	MOVZX ESI,WORD PTR DS:[EAX]	
01144079	C1C6 02	ROL ESI,0x2	
0114407C	03D6	ADD EDX,ESI	
0114407E	8B02	MOV EAX,DWORD PTR DS:[EDX]	wvs.00905A4D
01144080	03C7	ADD EAX,EDI	kernel32.75F00000
01144082	8B4C24 F8	MOV ECX,DWORD PTR SS:[ESP-0x8]	kernel32.75F00000
01144086	68 90471401	PUSH 0x1144790	ASCII "CreateThread"
01144088	51	PUSH ECX	kernel32.75F00000
0114408C	FFD0	CALL EAX	kernel32.CreateThread
0114408E	90	NOP	
0114408F	90	NOP	
01144090	90	NOP	
01144091	90	NOP	

Vaya ahora bien en código les explico los parámetros que debemos meter como lo hicimos anteriormente pero ahora para cargar nuestro hilo:

```
mov eax, offset proced
invoke CreateThread,NULL,NULL,eax,NULL,NULL,NULL
```

Todo es "NULL" o "0" y lo único que debe importarnos es ubicar en algún registro la dirección donde queremos poner nuestro hilo y quedaría algo así en el código injertado:

0114408C	FFD0	CALL EAX	kernel32.CreateThread
0114408E	B9 B0401401	MOV ECX,0x11440B0	
01144093	6A 00	PUSH 0x0	
01144095	6A 00	PUSH 0x0	
01144097	6A 00	PUSH 0x0	
01144099	51	PUSH ECX	kernel32.75F00000
0114409A	6A 00	PUSH 0x0	
0114409C	6A 00	PUSH 0x0	
0114409E	FFD0	CALL EAX	kernel32.CreateThread
011440A0	EB FE	JMP SHORT 011440A0	wvs.011440A0
011440A2	90	NOP	
011440A3	90	NOP	
011440A4	90	NOP	
011440A5	90	NOP	
011440A6	90	NOP	
011440A7	90	NOP	
011440A8	90	NOP	
011440A9	90	NOP	
011440AA	90	NOP	
011440AB	90	NOP	
011440AC	90	NOP	
011440AD	90	NOP	
011440AE	90	NOP	
011440AF	90	NOP	
011440B0	90	NOP	
011440B1	90	NOP	

Como verán meti en el registro ECX un offset que será donde ubicaremos nuestro loop para buscar código y será nuestro hilo... (ignoren el ultimo “JMP” lo ubique para probar si el hilo era tomado) por lo tanto con esto avanzado podríamos recuperar los valores de la pila a como estaban con la instrucción POPAD (recuerden que al inicio colocamos un pushad para salvar los registro en la pila) y regresar a donde originalmente iniciaba el programa y tendríamos un hilo corriendo en paralelo buscando los bytes a parchear que adelante mostrare, mientras pongamos el código restante para devolver la ejecución al inicio del programa como normalmente comenzaría:

011440A0	61	POPAD	
011440A1	E9 5AEFFFFF	JMP 01143000	wvs.01143000
011440A6	90	NOP	
011440A7	90	NOP	
011440A8	90	NOP	
011440A9	90	NOP	

Recuperamos los registros y ponemos un salto a donde iniciaba el programa antes de agregar la sección con topo, antes de guardar los cambios que ya hemos hecho, pasemos al offset donde se encontraría el hilo y metamos el código, pero antes recordemos algo en qué dirección se ubicaba el salto que queríamos cambiar (no tocare detalles de crackeo del software) , pero quiero mostrarles como luce cuando aún no es ejecutado el programa y el código está cifrado:

(ESTA NO ES LA LIMITACION DEL SOFTWARE, ES UN EJEMPLO)

Cifrado:

00462957	B1 82	MOV CL, 0x82	
00462959	9B	WAIT	
0046295A	39B6 C075C76D	CMP DWORD PTR DS:[ESI+0x6DC775C0],ESI	
00462960	79 B5	JNS SHORT 00462917	wvs.00462917
00462962	26:2231	AND DH, BYTE PTR ES:[ECX]	
00462965	3C 2C	CMP AL, 0x2C	
00462967	98	CWDE	
00462968	CA B311	RETF 0x11B3	Far return
0046296B	65:48	DEC EAX	Superfluous prefix
0046296D	B2 9A	MOV DL, 0x9A	
0046296F	F1	INT1	
00462970	32B6 63AE9269	XOR DH, BYTE PTR DS:[ESI+0x6992AE63]	
00462976	B8 AC27DA5F	MOV EAX, 0x5FDA27AC	
0046297B	14 F0	ADC AL, 0xF0	
0046297D	A4	MOVS BYTE PTR ES:[EDI], BYTE PTR DS:[ESI]	

Descifrado:

00462957	81B6	TEST BYTE	
00462959	74 60	JE SHORT 004629BB	wvs.004629BB
0046295B	6A 10	PUSH 0x10	
0046295D	6A 00	PUSH 0x0	
0046295F	8DAE F0FFFFFF	LEA EDI, DWORD PTR SS:[FEB-0x120]	
00462965	A1 F413AE00	MOV EAX, DWORD PTR DS:[0xAE13F4]	
0046296A	8B08	MOV ECX, DWORD PTR DS:[EAX]	
0046296C	FF 91 48020000	CALL DWORD PTR DS:[ECX+0x248]	
00462970	8DAE F0FFFFFF	LEA EDI, DWORD PTR SS:[FEB-0x120]	

Ahora bien, los bytes descifrados son los siguientes y por lo tanto los que deberíamos buscar en nuestro loop son los siguientes:

Address	Hex dump	ASCII
00462959	74 60 6A 10	6A 00 8D 95
00462961	E0 FE FF FF	A1 F4 13 AE
00462969	00 8B 08 FF	91 48 02 00
00462971	00 8B 85 E0	FE FF FF E8
00462979	DF 34 FA FF	50 6A 00 E8

Y cambiarlos por lo que muestro a continuación:

00462957	81B6	TEST DL, DL	
00462959	EB 60	JMP SHORT 004629BB	
0046295B	6A 10	PUSH 0x10	
0046295D	6A 00	PUSH 0x0	
0046295F	8DAE F0FFFFFF	LEA EDI, DWORD PTR SS:[FEB-0x120]	
004629BB			wvs.004629BB

Address	Hex dump	ASCII
00462959	EB 60 6A 10 6A 00 8D 95	6A 00 8D 95
00462961	E0 FE FF FF A1 F4 13 AE	A1 F4 13 AE
00462969	00 8B 08 FF 91 48 02 00	91 48 02 00

Como ven se marcan en rojo los cambios que hice cambie el salto JE (condicional) por un JMP para que salte siempre sin excepción, solo cambia un byte (EB) con el parche que queremos colocar, por lo tanto una vez encontrado en memoria cuando se ha descifrado el programa parchearemos ese byte y colocaremos “EB” y listo.

Ahora escribamos el código siguiente:

011440B8	90	NOP	
011440B9	81 3D 59294600	CMP DWORD PTR DS:[0x462959],0x106A6074	
011440BA	75 F4	JNZ SHORT 011440B0	wws.011440B0
011440BC	C7 05 59294600	MOV DWORD PTR DS:[0x462959],0x106A60EB	
011440C6	EB FE	JMP SHORT 011440C6	wws.011440C6
011440C8	90	NOP	
011440C9	90	NOP	
011440CA	90	NOP	

Como vimos hace una búsqueda de los bytes y sigue repitiéndose el loop hasta que los encuentra y pasa guardar lo bytes nuevos en esa dirección que como verán solo cambia el “EB” y mas abajo coloque un loop infinito (solo para probar lo que ya hemos hecho, si dejamos esto asi el loop infinito consumiría muchos recursos) podríamos hacer mas cosas como cerrar el hilo, poner un sleep o demas cosas, el tema es que ya terminamos de hacer toda la máquina de un inline patching y esto era básicamente lo que quería mostrarles! Y ojala que les sirva de algo esto ya que me tomo muchas horas armarlo ☺.

aquí les dejo el proyecto: <http://uppit.com/4r8bnmh4fxrc/OBTENER-DLL.rar>

Antes de terminar, les cuento que hace un rato hable con guan de dio y le platique la idea de hacer una tool con un método “general” para hacer inline patching a themida, osea automatizar todo esto que hice a mano en una herramienta que básicamente haga lo siguiente: nos pida offset y bytes nuevos a colocar en la dirección y listo haría todo lo que tarde horas en 2 segundos, creo que podría ser una excelente herramienta para inlinear a themida, cabe mencionar que ya he probado el código con diferentes sistemas operativos y repare detalles que causaban problemas, sin mas les mando un saludo y si alguien se quiere unir al proyecto espero sus comentarios! , saludos a todos!

Alejandro (torrescrack)

Tora_248@hotmail.com