

## Ofuscación de malware

### Descripción de las herramientas necesarias para el análisis de malware

#### PEBear

Es una herramienta para analizar de forma estática ejecutables para Windows. Tiene una ventaja sobre los otros programas similares que he probado, y es que es mas visual que el resto. Además, permite editar directamente el archivo. Esto es importante porque, hay veces, que tenemos que arreglar la tabla PE para poder ejecutar el sample correctamente.

#### X64dbg

Como descrito en la practica anterior, es un debugger de ejecutables. Con este debugger podemos hacer análisis estático y análisis dinámico. Además, se puede editar el archivo, para añadir comentarios, para poder hacer que la practica de reversing sea mas sencilla. Estos cambios, no se guardan directamente en el binario, sino que se guardan en una base de datos, que al detectar el binario, añade los comentarios, puntos de ruptura y trabajo que hayas realizado con el ejecutable. Tiene una ventaja sobre IDA, por ejemplo, y es que cuenta con la herramienta Scylla de fabrica.

#### Scylla

## Prueba 1

### Análisis estático de la prueba de malware descargada

Abrimos el malware descargado con PEBear para ver que efectivamente, se trata de un programa para Windows. Este ejecutable, como la mayoría de malwares, esta compilado para una arquitectura de 32 bits. De esta forma, tiene mas posibilidades de correr en distintas maquinas.

### Análisis desde el punto de vista del empaquetado de ejecutables (Un binario distinto al inicial)

En esta sección se pretende desempaquetar el malware seleccionado para posteriormente hacer un volcado de memoria virtual a disco y averiguar las cabeceras. De esta forma, podemos analizar el malware como se ha creado desde el inicio, sin los inconvenientes introducidos por los packers.

De primeras, vemos que el ejecutable que estamos analizando no esta packed con ningún malware. Sabemos que el malware es autocontenido porque el perfil del mismo en VirusShare, vemos que no crea ningún hilo de ejecución. Esto quiere decir que tenemos que estar atentos a las funciones: `Kernel32::VirtualAlloc()` y `Kernel32::VirtualProtect()`.

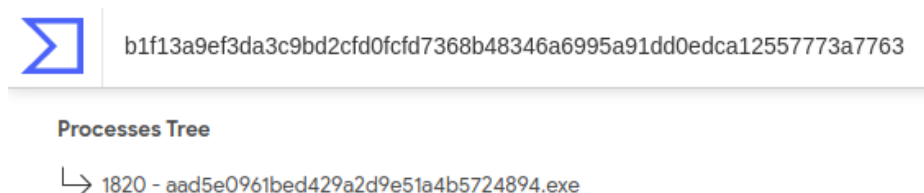


Figure 1: Process Tree del malware sample

Efectivamente, siguiendo la sección de memoria que se copula, vemos que se inserta un archivo PE entero. Volcando el contenido de memoria a disco, vemos que el malware esta packed. Además, podemos determinar que las cabeceras están sin mapear, que es como las espera Windows, por tanto, podemos directamente ir a desempaquetar el malware. La herramienta que se ha usado para el empaquetado del mismo es UPX, una herramienta muy conocida en el mundo de los packers.

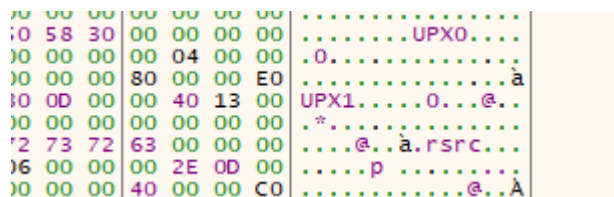


Figure 2: Malware Packed volcado en memoria

### Desempaquetado automático

Para hacer el unpack, vamos a usar la misma herramienta que se usa para packear, ya que si se le pasa la opción -d, el packer te unpackea cualquier archivo que ha packeado el programa.

```
C:\Users\windows1\Desktop\Malware\Shade-Ransom>upx -d Shade_02250000.bin
          Ultimate Packer for eXecutables
          Copyright (C) 1996 - 2020
UPX 3.96w      Markus Oberhumer, Laszlo Molnar & John Reiser   Jan 23rd 2020

-----
File size      Ratio      Format      Name
-----
2079232 <-    868352    41.76%    win32/pe    Shade_02250000.bin
Unpacked 1 file.
```

Figure 3: UPX -d desempaquetado

En este momento, tenemos el malware unpacked y ya podemos pasar a analizarlo.

## Desempaquetado manual

Se ha decidido usar X64dbg para hacer el desempaquetado manual. Para poder ejecutar un programa (Malware, en este caso) tiene que estar descomprimido. Los packers son útiles para evadir el malware de los programas automatizados de control de malware, pero cuando tenemos pruebas de que un malware ha sido packeado, sabemos que vamos a poder extraerlo sin mucho apuro, puesto que en algún momento, este malware va a tener que ser descomprimido y copiado a una sección de memoria, para poder pasarle el control al mismo. De esta forma, tenemos que buscar en el código del packer, una llamada a una función fuera de la zona de memoria actual.

El procedimiento es el siguiente:

1. Poner un punto de ruptura de acceso en la zona UPX0
  - Esto lo hacemos porque cuando se ceda el control a esta zona de memoria, ya estará el malware descomprimido.
  - En el caso en el que se nos escape la llamada a la zona de memoria, este punto de ruptura evitara que el malware se ejecute en nuestro sistema.
2. Analizar el código hasta que se entrega el control a la zona de memoria fuera de la sección UPX1, que es la del código “Stub”.
3. Poner el breakpoint justo antes de esa llamada y sacar el binario de la zona de memoria ya poblada con el malware descomprimido.
4. Si el malware, no tiene las cabeceras, usar Scylla para arreglar la tabla de Imports y volcar la zona a disco como un PE “Standalone”

00400000	00001000	shade_02370000.exe		IMG	-R---	ERWC-
00534000	00003000	"UPX0"		IMG	ERW-G	ERWC-
00534000	00003000	"UPX1"		IMG	ERWC-	ERWC-
00607000	00001000	".rsrc"	Resources	IMG	-RW--	ERWC-

Figure 4: Punto de Ruptura en UPX0

Si entramos a revisar el contenido de la zona de memoria asignada a UPX0, vemos que no hay nada escrito en esta zona de memoria. Esto puede ser un indicativo de que cuando el código acabe de descombrar el malware, este sera guardado en esta zona.

Si observamos la siguiente foto, podemos ver como ahora, la zona de memoria que estaba completamente vacía, esta siendo poblada con código.

Llegados a este punto, sabemos también que al ser una aplicación de consola, Windows va a llamar a la función `GetCommandLineA` así que le asignamos un breakpoint a esta también.

Corremos el código hasta que encontramos que se le llama, seguimos el trazo hasta que devuelve. Un poco mas abajo, vemos una llamada que hace referencia a la dirección de memoria que se ejecuta con el malware descomprimido. Se puede ver en la siguiente foto.

●	00401000	0000	add byte ptr ds:[eax],al
●	00401002	0000	add byte ptr ds:[eax],al
●	00401004	0000	add byte ptr ds:[eax],al
●	00401006	0000	add byte ptr ds:[eax],al
●	00401008	0000	add byte ptr ds:[eax],al
●	0040100A	0000	add byte ptr ds:[eax],al
●	0040100C	0000	add byte ptr ds:[eax],al
●	0040100E	0000	add byte ptr ds:[eax],al
●	00401010	0000	add byte ptr ds:[eax],al
●	00401012	0000	add byte ptr ds:[eax],al
●	00401014	0000	add byte ptr ds:[eax],al
●	00401016	0000	add byte ptr ds:[eax],al
●	00401018	0000	add byte ptr ds:[eax],al
●	0040101A	0000	add byte ptr ds:[eax],al
●	0040101C	0000	add byte ptr ds:[eax],al
●	0040101E	0000	add byte ptr ds:[eax],al
●	00401020	0000	add byte ptr ds:[eax],al
●	00401022	0000	add byte ptr ds:[eax],al
●	00401024	0000	add byte ptr ds:[eax],al
●	00401026	0000	add byte ptr ds:[eax],al
●	00401028	0000	add byte ptr ds:[eax],al
●	0040102A	0000	add byte ptr ds:[eax],al
●	0040102C	0000	add byte ptr ds:[eax],al
●	0040102E	0000	add byte ptr ds:[eax],al
●	00401030	0000	add byte ptr ds:[eax],al
●	00401032	0000	add byte ptr ds:[eax],al
●	00401034	0000	add byte ptr ds:[eax],al

Figure 5: UPX0 sin datos

●	00401000	56	push esi
●	00401001	8BF1	mov esi,ecx
●	00401003	E8 71000018	call 18401079
●	00401008	F64424 08 01	test byte ptr ss:[esp+8],1
●	0040100D	74 07	je shade_02370000.401016
●	0040100F	56	push esi
●	00401010	E8 7114C80C	call 0082486
●	00401015	59	pop ecx
●	00401016	8BC6	mov eax,esi
●	00401018	5E	pop esi
●	00401019	C2 0400	ret 4
●	0040101C	56	push esi
●	0040101D	8BF1	mov esi,ecx
●	0040101F	8B46 08	mov eax,dword ptr ds:[esi+8]
●	00401022	C706 80B45800	mov dword ptr ds:[esi],shade_02370000.5
●	00401028	85C0	test eax,eax
●	0040102A	74 15	je shade_02370000.401041
●	0040102C	50	push eax
●	0040102D	FF76 04	push dword ptr ds:[esi+4]
●	00401030	FF15 8CF95F00	call dword ptr ds:[5FF98C]
●	00401036	85C0	test eax,eax
●	00401038	74 07	je shade_02370000.401041
●	0040103A	50	push eax
●	0040103B	FF15 90F95F00	call dword ptr ds:[5FF990]
●	00401041	FF76 04	push dword ptr ds:[esi+4]
●	00401044	FF15 94F90000	call dword ptr ds:[F994]
●	0040104A	0000	add byte ptr ds:[eax],al
●	0040104C	0000	add byte ptr ds:[eax],al
●	0040104E	0000	add byte ptr ds:[eax],al
●	00401050	0000	add byte ptr ds:[eax],al
●	00401052	0000	add byte ptr ds:[eax],al
●	00401054	0000	add byte ptr ds:[eax],al
●	00401056	0000	add byte ptr ds:[eax],al
●	00401058	0000	add byte ptr ds:[eax],al
●	0040105A	0000	add byte ptr ds:[eax],al
●	0040105C	0000	add byte ptr ds:[eax],al
●	0040105E	0000	add byte ptr ds:[eax],al
●	00401060	0000	add byte ptr ds:[eax],al
●	00401062	0000	add byte ptr ds:[eax],al

Figure 6: UPX0 siendo poblada

```

0054E45F E8 364F0000 call shadePhase1.55339A
0054E464 85C0 test eax, eax
0054E466 7D 08 jge shadePhase1.54E470
0054E468 6A 18 push 18
0054E46A E8 99270000 call shadePhase1.550C08
0054E46F 59 pop ecx
0054E470 FF15 90405800 call dword ptr ds:[<&GetCommandLineA>]
0054E476 A3 24176000 mov dword ptr ds:[601724], eax
0054E478 E8 E3400000 call shadePhase1.553263
0054E480 A3 808A5F00 mov dword ptr ds:[5F8A80], eax
0054E485 E8 1E4D0000 call shadePhase1.5531A8
0054E48A 85C0 test eax, eax
0054E48C 7D 08 jge shadePhase1.54E496
0054E48E 6A 08 push 8
0054E490 E8 73270000 call shadePhase1.550C08
0054E495 59 pop ecx
0054E496 E8 954A0000 call shadePhase1.552F30
0054E49B 85C0 test eax, eax
0054E49D 7D 08 jge shadePhase1.54E4A7
0054E49F 6A 09 push 9
0054E4A1 E8 62270000 call shadePhase1.550C08
0054E4A6 59 pop ecx
0054E4A7 53 push ebx
0054E4A8 E8 1A280000 call shadePhase1.550CC7
0054E4AD 59 pop ecx
0054E4AE 3BC6 cmp eax, esi
0054E4B0 74 07 je shadePhase1.54E4B9
0054E4B2 50 push eax
0054E4B3 E8 50270000 call shadePhase1.550C08
0054E4B8 59 pop ecx
0054E4B9 E8 134A0000 call shadePhase1.552ED1
0054E4BE 845D C4 test byte ptr ss:[ebp-3C], b1
0054E4C1 74 06 je shadePhase1.54E4C9
0054E4C3 0FB74D C8 movzx ecx, word ptr ss:[ebp-38]
0054E4C7 EB 03 jmp shadePhase1.54E4CC
0054E4C9 6A 0A push A
0054E4CB 59 pop ecx
0054E4CC 51 push ecx
0054E4CD 50 push eax
0054E4CE 56 push esi
0054E4CF 68 00004000 push shadePhase1.400000
0054E4D4 E8 BF3EECF mov dword ptr ss:[ebp-20], eax
0054E4D9 8945 E0 cmp dword ptr ss:[ebp-1C], esi
0054E4DC 3975 E4 jne shadePhase1.54E4E7
0054E4DF 75 06 jbe shadePhase1.54E4E7

```

Figure 7: Llamada al OEP

En este momento, tenemos el OEP. Ejecutamos hasta que llegamos al primer breakpoint, hacemos un **step into** y abrimos la dirección a la que apunta nuestro **eip** con Scylla.

Hacemos un dump de memoria para luego abrirlo y hacer un PE rebuild. Esto sirve para reconstruir la table PE. Sin esto, no podemos abrir el ejecutable de forma independiente.

## Análisis de malware desde el punto de vista de ofuscación

Una vez tenemos el Malware desempaquetado, podemos ver que hace dos llamadas en las primeras líneas de código a una dirección de memoria, sin nada asignado. Esto, en nuestro caso significa que el malware construye su IAT de forma dinámica. Para poder analizar el código bien, tenemos que averiguar que función carga las librerías de forma dinámica.

Al seguir el flujo, vemos que los nombres de las librerías que se cargan están cifrados. Esto lo sabemos porque esta usando la función **GetProcAddress** y si miramos en la documentación de Microsoft, vemos que se le tiene que pasar el nombre de la función o variable.

Vamos a proceder a descifrar los nombres. Para hacer esto, lo mas sencillo es encontrar la función que descifra los nombres y dejar que se ejecute en un análisis dinámico. Esto nos dejaría con todos los nombres de los métodos importados resueltos.

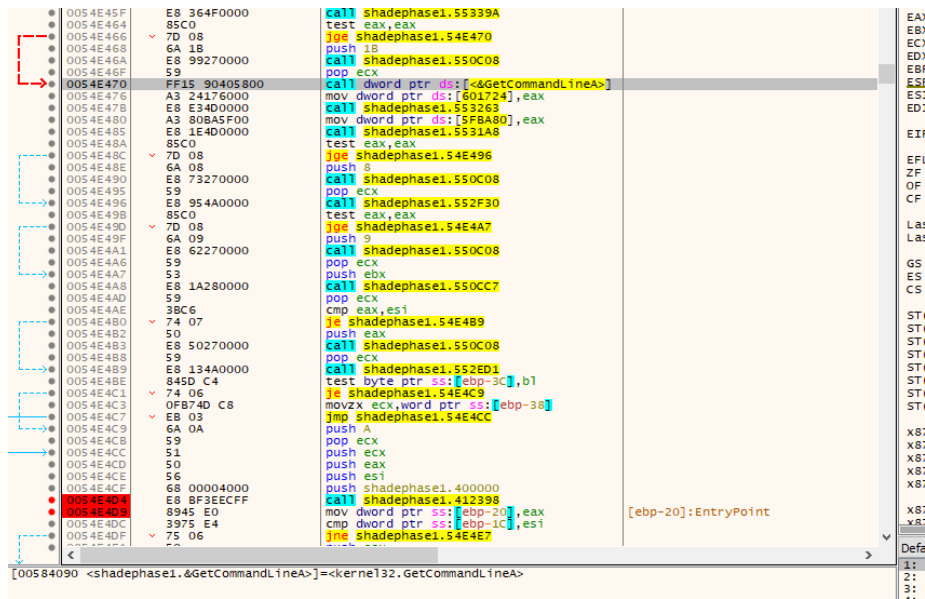


Figure 8: Proceso de encontrar el OEP

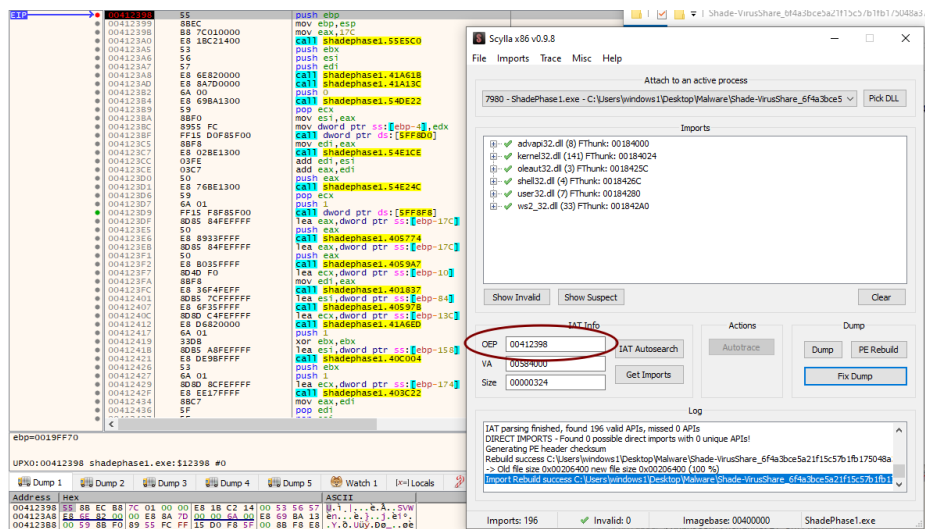


Figure 9: Volcado del malware unpacked a disco

```

push    edi
call    sub_41A61B
call    sub_41A13C
push    0
call    sub_54DE22
pop     ecx
mov     esi, eax
mov     [ebp+var_4], edx
call    ds:dword_5FF8D0
mov     edi, eax
call    sub_54E1CE
add     edi, esi
add     eax, edi
push    eax
call    sub_54E24C
pop     ecx
push    1
call    ds:dword_5FF8F8
lea     eax, [ebp+var_17C]
push    eax
call    sub_405774


```

Figure 10: Prueba de IAT

xrefs to dword_5FF8D0			
Direction	Type	Address	Text
Up	r	sub_410FF9+55	call ds:dword_5FF8D0
Up	r	sub_411F32+7B	call ds:dword_5FF8D0
	r	start+27	call ds:dword_5FF8D0
D...	r	sub_4151EA+5C	call ds:dword_5FF8D0
D...	r	sub_415DB8+18	call ds:dword_5FF8D0
D...	w	sub_41A13C+166	mov ds:dword_5FF8D0, eax
D...	r	sub_43CFC9+3C	call ds:dword_5FF8D0

Figure 11: Referencias a la zona de memoria reservada

## Syntax

C++	 Copy
<pre>FARPROC GetProcAddress(     HMODULE hModule,     LPCSTR lpProcName );</pre>	

## Parameters

`hModule`

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`

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.

## Return value

If the function succeeds, the return value is the address of the exported function or variable.

If the function fails, the return value is NULL. To get extended error information, call [GetLastError](#).

Figure 12: Documentación GetProcAddress



```

mov     ds:dword_5FF8BC, eax
push    offset aQ      ; "(q"
push    edi            ; hModule
call    esi ; GetProcAddress
push    offset byte_5F8EDC ; lpProcName
push    edi            ; hModule
mov     ds:dword_5FF8C0, eax
call    esi ; GetProcAddress
push    offset byte_5F8EFC ; lpProcName
push    edi            ; hModule
mov     ds:dword_5FF8C4, eax
call    esi ; GetProcAddress
push    offset byte_5F8F0C ; lpProcName
push    edi            ; hModule
mov     ds:dword_5FF8C8, eax
call    esi ; GetProcAddress
push    offset aCd      ; "]/Cd"
push    edi            ; hModule
mov     ds:dword_5FF8CC, eax
call    esi ; GetProcAddress
push    offset aG      ; "g"
push    edi            ; hModule
mov     ds:dword_5FF8D0, eax
call    esi ; GetProcAddress
push    offset byte_5F8F4C ; lpProcName
push    edi            ; hModule
mov     ds:dword_5FF8D4, eax
call    esi ; GetProcAddress
push    offset byte_5F8F5C ; lpProcName
push    edi            ; hModule
mov     ds:dword_5FF8D8, eax
call    esi ; GetProcAddress
push    offset byte_5F8F7C ; lpProcName
push    edi            ; hModule
mov     ds:dword_5FF8DC, eax

```

Figure 13: Prueba de que existe texto cifrado y algoritmo de descifrado

No se ha averiguado el tipo de cifrado que ha sido implementado en el malware.

### Búsqueda de cifrado Cesar

En el malware, no se ha encontrado ninguna cadena que haga referencia al cifrado Cesar.

### Búsqueda de cifrado Base64

Se han buscado las cadenas que posiblemente puedan contener la clave del cifrado Base64, pero tampoco se ha encontrado nada. El patrón que se ha buscado es el siguiente:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

Al ser una parte de la cadena, si se encuentra esta cadena, seguramente encontremos la clave del cifrado Base64. Tampoco se ha encontrado ninguna ocurrencia de la cadena.

### Búsqueda de cifrado XOR

Para encontrar las funciones de cifrado XOR hemos buscado con IDA XOR en el buscador. Esto nos ha sacado muchas instrucciones (14839), pero solo 18 funciones han resultado tener XOR que no comparen el mismo registro. Se ha empezado a analizar la función `sub_532537` y se ha encontrado una estructura que tiene un símil muy cercano al de una función `xor` sencilla.

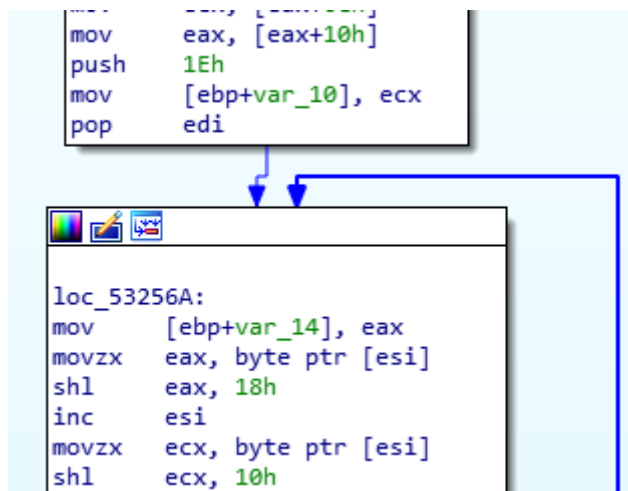


Figure 14: Cifrado XOR parte superior

Analizando el malware para buscar funciones con la esta

```

xor     eax, [ebp+var_28]
push    [ebp+var_14]
mov     [ebp+var_38], eax
call    sub_56F4C8
mov     ecx, [ebp+var_18]
xor     ecx, [ebp+var_4]
add     eax, [ebp+var_C]
xor     ecx, [ebp+var_8]
push    edi
add     ecx, [ebp+var_38]
push    [ebp+var_18]
lea     eax, [ecx+eax-359D3E2Ah]
mov     [ebp+var_10], eax
call    sub_56F4C8
mov     [ebp+var_18], eax
mov     eax, [ebp+var_50]
xor     eax, [ebp+var_4C]
push    ebx
xor     eax, [ebp+var_40]
push    [ebp+var_10]
xor     eax, [ebp+var_1C]
mov     [ebp+var_40], eax
call    sub_56F4C8
mov     ecx, [ebp+var_18]
xor     ecx, [ebp+var_14]
add     eax, [ebp+var_4]
xor     ecx, [ebp+var_8]

```

Figure 15: Cifrado XOR parte intermedia

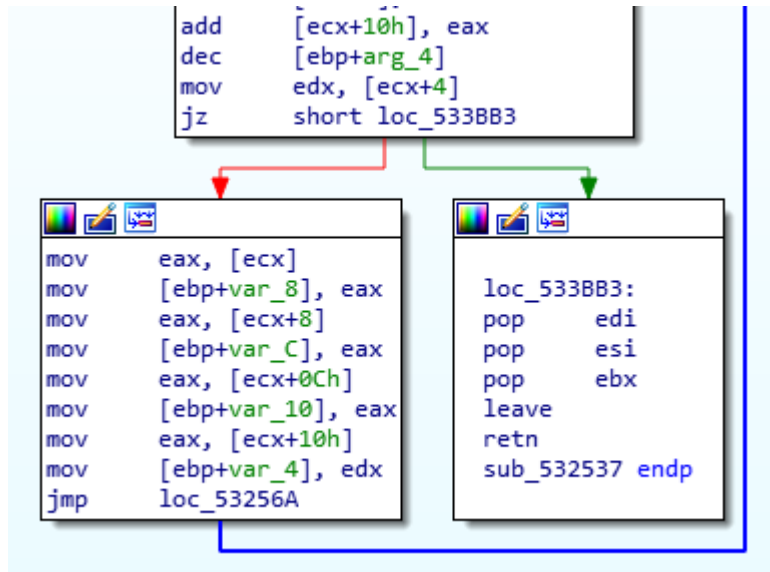


Figure 16: Cifrado XOR parte inferior

## Prueba 2

### Análisis Estático

Con el análisis de las cabeceras, podemos ver que es un ejecutable, porque tiene la cabecera mágica. Además, observando el tamaño del RAW Size de la sección .text y el tamaño del virtual size, vemos que existe una diferencia importante.

c41cbad1ee87b9156c389962608cf							
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocat
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word
.text	00002C10	00001000	00003000	00001000	00000000	00000000	0000
.rdata	00015FE8	00004000	00016000	00004000	00000000	00000000	0000
.data	000011B8	0001A000	00001000	0001A000	00000000	00000000	0000
.rsrc	00007170	0001C000	00008000	0001B000	00000000	00000000	0000
.reloc	000000E0	00024000	00001000	00023000	00000000	00000000	0000

Figure 17: Cabeceras

### Análisis Dinámico

Al ver que es una muestra que está packed, podemos ejecutar la muestra con nuestro debugger hasta que se llame al proceso VirtualAlloc del API

Kernel32.dll.

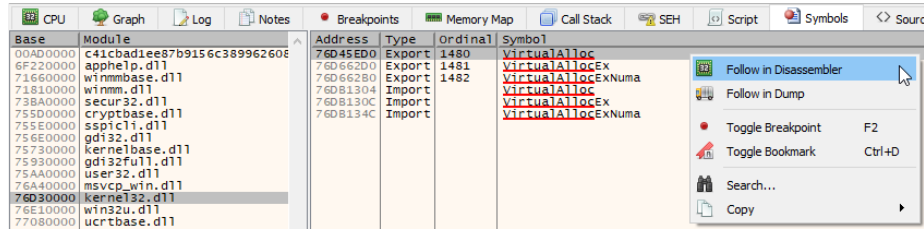


Figure 18: VirtualAlloc Breakpoint

Hacemos esto porque sabemos que el virus es autocontenido y que no crea un nuevo proceso desde si mismo. Esto quiere decir, que en algún momento, se debe reservar memoria para el virus.

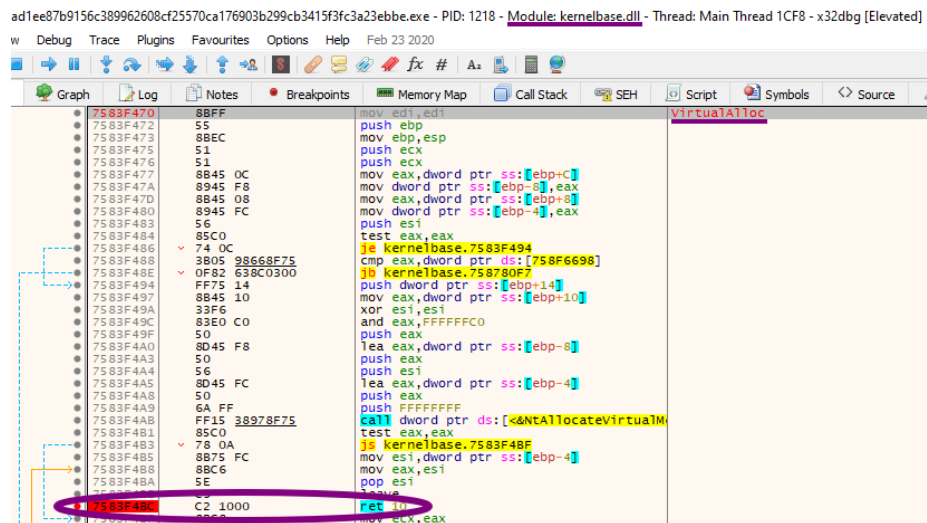


Figure 19: VirtualAlloc return de KernelBase

En este momento, tenemos los siguientes Breakpoints:

Según la documentación de Microsoft, podemos saber que este método devuelve el puntero a memoria en el registro EAX, entonces, basta con seguir el registro en el Dump para obtener el programa **Unpacked**. Tras dos iteraciones, se ha generado un archivo PE en una sección de memoria.

En este momento, sabemos que se ha guardado el archivo unpacked en memoria. Para llegar a el, tenemos que observar las direcciones de memoria que tienen el bit de ejecutable habilitado. En este sample en concreto, tenemos dos. Siguiendo la primera en el dump, podemos ver que esta en esta dirección nuestro PE. Para

Type	Address	Module/Label/Exception	State	Disassembly	Hits	Summary
Software	00AD16F0 7583F48C	<c41cbad1ee87b9156c3895 kernelbase.dll	One-time Enabled	xchg ebp,eax ret 10	0 0	entry breakpoint

Figure 20: Breakpoints

Address	Hex	ASCII
00AB0000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....yy..
00AB0010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	.....@.....
00AB0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....D.....
00AB0030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00AB0040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	...!.Li!Th
00AB0050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00AB0060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00AB0070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode...\$. ....
00AB0080	78 98 95 E3 3C FA FB 80 3C FA FB 80 3C FA FB 80	x...a<uú* <uú* <uú*
00AB0090	3C FA FB 80 20 FA FB 80 35 82 68 80 35 FA FB 80	<uú* uú*5.h*5uú*
00AB00A0	3C FA FB 80 3D FA FB 80 31 A8 25 80 3D FA FB 80	<uú*=uú*1%*=uú*
00AB00B0	52 69 63 68 3C FA FB 80 00 00 00 00 00 00 00 00	Rich<uú*.....
00AB00C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00AB00D0	50 45 00 00 4C 01 04 00 5A 88 5F 5A 00 00 00 00	PE..L..Z..Z..
00AB00E0	00 00 00 00 E0 00 02 01 08 01 0C 00 80 29 00 00	...ä.....*)..
00AB00F0	80 2A 00 00 00 00 00 00 02 39 00 00 00 10 00 00	.*.....9.....
00AB0100	00 40 00 00 00 00 40 00 00 10 00 00 10 00 00 00	.@.....e.....
00AB0110	05 00 00 00 00 00 00 00 05 00 00 00 00 00 00 00	.....
00AB0120	00 E0 00 00 70 02 00 00 00 00 00 00 03 00 40 81	.a..p.....@..

Figure 21: PE generado

extraer el sample unpacked, tenemos que volcar el contenido de la memoria en disco, como en la siguiente imagen:

Address	Size	Info	Content	Type	Protection	Initial
00AA0000	00004000			MAP	-R---	-R---
00AA4000	00004000	Reserved (00AA0000)		MAP	-R---	-R---
00AB0000	0000E000			PRV	ERW---	ERW---
00AC0000	00004000			MAP	-R---	-R---

Address	Hex	ASCII
00AB0000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....yy..
00AB0010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	.....@.....
00AB0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....D.....
00AB0030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00AB0040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	...!.Li!Th
00AB0050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00AB0060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00AB0070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode...\$. ....
00AB0080	78 98 95 E3 3C FA FB 80 3C FA FB 80 3C FA FB 80	x...a<uú* <uú* <uú*
00AB0090	3C FA FB 80 20 FA FB 80 35 82 68 80 35 FA FB 80	<uú* uú*5.h*5uú*

Figure 22: PE encontrado en memoria y volcando

Una vez tenemos el programa ejecutable en disco, lo volvemos a analizar con un editor PE.

Al abrirlo y analizar la sección .text que contiene el código ejecutable, vemos que hace referencia a una sección vacía. Esto es debido a que las cabeceras PE están cambiadas para apuntar a las direcciones de las secciones en memoria, no en disco. Debemos arreglar esto antes de poder abrir el PE con un debugger. Para hacer esto, lo primero que tenemos que hacer es cambiar las direcciones

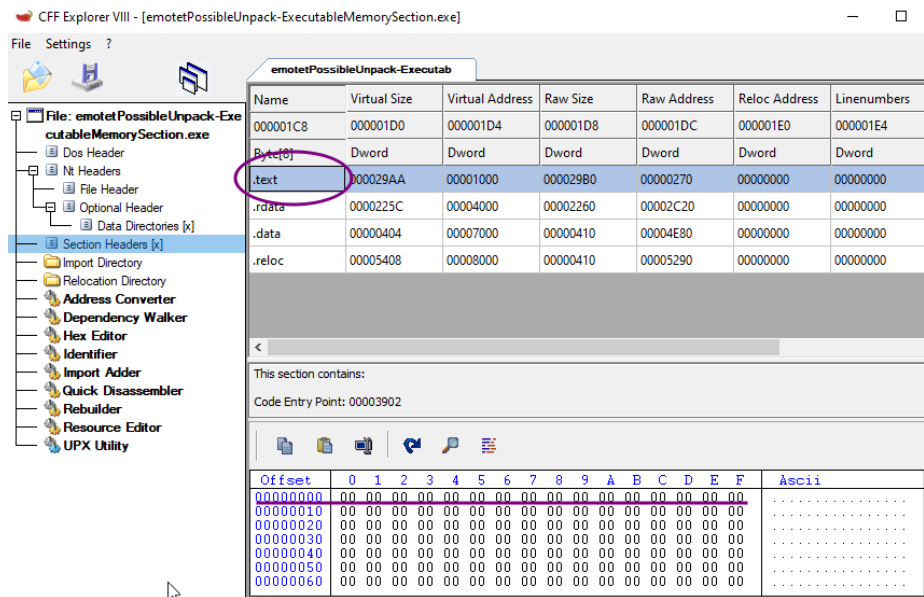


Figure 23: Cabeceras del binario mapeadas mal

referenciadas en **RawAddress** por las referenciadas en **VirtualAddress**, de forma que ambos campos tengan los mismos valores.

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocat
000001C8	000001D0	000001D4	000001D8	000001DC	000001E0	000001E4	000001
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word
.text	000029AA	00001000	000029B0	00001000	00000000	00000000	0000
.rdata	0000225C	00004000	00002260	00004000	00000000	00000000	0000
.data	00000404	00007000	00000410	00007000	00000000	00000000	0000
.reloc	00005408	00008000	00000410	00008000	00000000	00000000	0000

Figure 24: Cambio de valores de la tabla de cabeceras

Lo segundo que tenemos que hacer es cambiar la dirección base de la tabla de cabeceras opcionales para que coincida con la dirección de entrada del programa.

Una vez hecho esto, podemos abrir el malware para analizar el código malicioso sin distracciones.

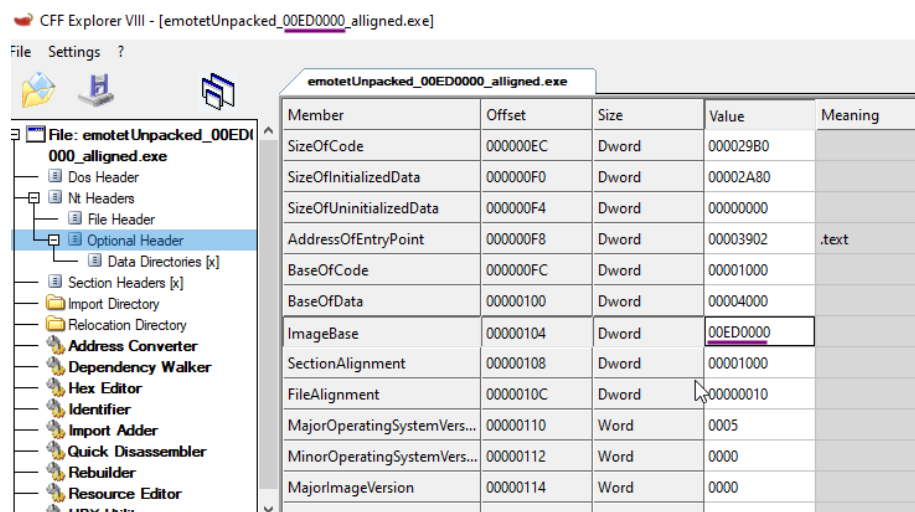


Figure 25: Cambio de dirección image base

## Análisis de malware desde el punto de vista de ofuscación (Con un malware distinto)

### Búsqueda de cifrado Cesar

En el siguiente malware, vamos a buscar indicios de que existe una implementación de cifrado Cesar

Se han buscado las cadenas del archivo y no se reconoce ninguna cadena potencialmente similar a la que sacaría un cifrado Cesar. Esto quiere decir que no es muy probable que se use un cifrado cesar en este malware.

### Búsqueda de cifrado Base64

Para averiguar si se usa Base64, vamos a buscar en todas las cadenas del malware una que contenga todos los caracteres que se usan para cifrar con Base64. Es cierto, que a veces, los autores de malware, usan una cadena que han decidido ellos, pero la idea de que esta tiene que contener todos los valores posibles, persiste.

Buscando en este malware la cadena que contiene los valores posibles Base64 o la cadena parcial, pero no se encuentra.

### Búsqueda de cifrado XOR

Iniciamos la búsqueda de cadenas ofuscadas. Empezamos con la técnica de ofuscación XOR. Buscando por Seringa en IDA, vemos bastantes resultados con XOR, unos 112, para ser exactos, pero de estos, los que nos interesan, son solo



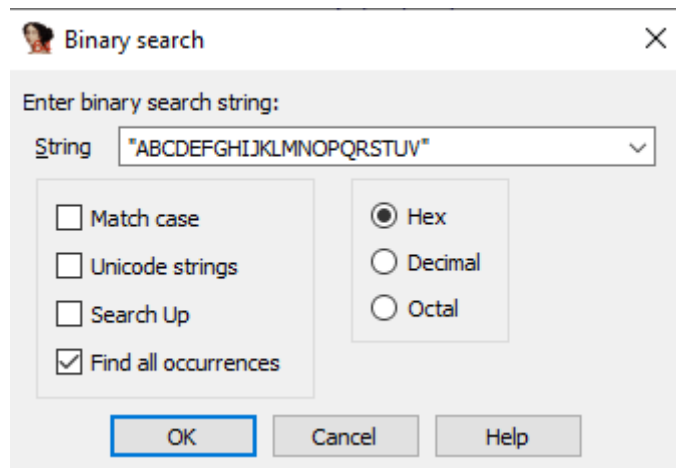


Figure 26: Búsqueda de cifrado por Base64

unos 11. Todos estos están en la misma función, que parece que sea una función de cifrado XOR. Los XOR que nos interesan son los que no realizan la misma operación sobre el registro, ya que esto, en un XOR siempre devuelve 0.

## Descripción de las herramientas necesarias para el análisis de malware

### PEBear

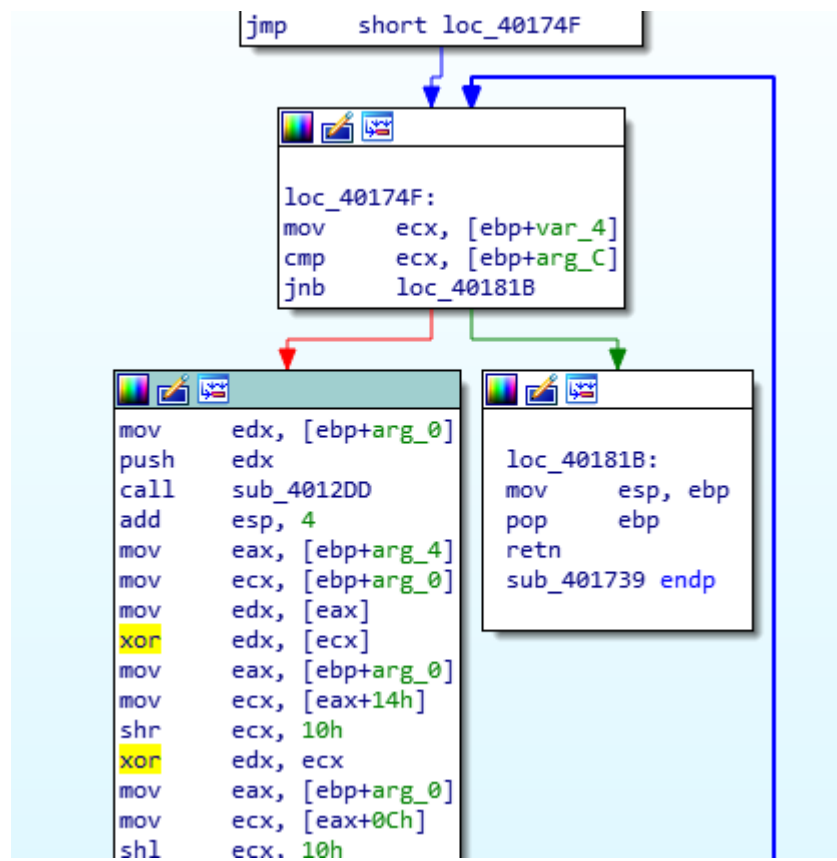


Figure 27: Función XOR de cifrado