

SOLUCIÓN CRACKME HISPASEC

1 – UNPACKING del TARGET

El primer requisito del reto, es desempaquetar el ejecutable. Vamos allá. **PeID** con firmas actualizadas de Unpack.CN, nos dice que está protegido con "**PESpin 0.3x - 1.xx -> cyberbob**". PeID se basa en signatures de bytes para detectar packers y no siempre es fiable, pero (por lo analizado posteriormente) en este caso ha dado en el clavo.

Empezamos a tracear para buscar el **OEP** y vemos que el código es muy aburrido, con junk code para marear. Vamos a ir un poquito más a saco. Ponemos **HW breakpoint** en **00401000h** y salta varias veces hasta q vamos a parar a un **REP MOVSB** en **0040A7C1h**. Tras este copiado de bytes tenemos .text desenscriptado. El siguiente acceso a .text, teóricamente, debería ser ya por ejecución. Así que vamos a la pestaña de "Memory" de Ollydbg y ponemos un "Set memory breakpoint on access" en la primera sección (**401000h-403000h**).

F9 y rompemos en **00401280h**. Vemos el siguiente código:

```
00401280  push  ebp
00401281  mov   ebp, esp
00401283  sub   esp, 8
00401286  mov   dword ptr [esp], 1
0040128D  call  __set_app_type
00401293  call  sub_401150
```

"**__set_app_type**" es un inicio característico de programas que usan "msvcrt.dll". Para el estudio del crackme bajé "PESpin" y vi que entre otras opciones tenía la de arrancar instrucciones del OEP. Pero si miramos la pila, vemos que la dirección de retorno es un address de Kernel32, el "**CALL [EBP+8]**" con el que XP salta al EntryPoint de todos los programas.

Es momento, de hacer un dump a disco. Ahora faltaría corregir los imports. En el OEP nos fijamos en el "**__set_app_type**" y vemos que en realidad es un "**CALL [00406100h]**". Vemos esa zona de memoria desde Olly y subimos hasta lo que parece el principio de los thunks, **004060F4h**. Le decimos a **Imprec** esta address, calculamos el size (54h) y le damos a "**Get Imports**". Nos llama la atención que sólo hay llamadas a "**msvcrt.dll**". Corregimos el dump (con **Imprec**), ejecutamos el crackme y va. En teoría good no? Pero si lo ejecutamos en otro PC el crackme peta. Nos hemos dejado algo sin corregir.

Puesto que PESpin dispone de macros de Encrypt/Decrypt en runtime, pensé que podía ser eso, pero tras analizar el desensamblado de .text no vi nada raro, así que es tema de imports. Bien, vamos a ponernos el mono de trabajo y mirar de arreglar los imports a mano (como a la vieja usanza). Via "**LoadLibraryA**" rompemos en el loader de imports. Vemos cómo desenscripta el nombre de las DLL's y en qué offsets están los nombres de estas DLL's. Sabiendo el offset, buscamos DWORD's que apunten a esa dirección y encontramos los **IMAGE_IMPORT_DESCRIPTOR** de la Import Table. Hasta aquí muy bien, pero ahora viene lo no tan guay. En la Import Table original no están los nombres de las funciones. En su lugar PESpin pone un BYTE con la primera letra de la string del import y a continuación un DWORD con **CRC32 custom** (al estilo viri ;) de la string del nombre de la API. Lo de la primera letra del import lo hace para hacer sólo el CRC de las API's que empiecen por esa letra y ganar en velocidad.

Bueno, sabiendo el address de la API que llama podríamos coger el nombre y meterlo ahí a mano, pero acabamos antes analizando el desensamblado y viendo dónde están los imports que faltan. Concretamente están en **0040BE60h**. La "putadilla" es que cada thunk está separado por un BYTE y hay que irle diciendo uno por uno a **Imprec**. Ahora sí, corregimos el

dump y todo funciona OK. Vamos a por el keygen ;p

2 – ESTUDIO y KEYGEN del TARGET

Empezamos a tracear en el OEP y vemos que hasta **00401E63h** todo son rutinas de inicialización del compilador. Ya en **00401E63h**, podemos ver varios "printf" y "scanf" que se encargan de la salida/entrada de la consola de msdos.

Umm lo raro es que hasta que no pasamos "cerca" de ellos no vemos la string que va a imprimir... Si nos fijamos antes y después de cada printf el programa hace una llama a **00401851h**. Esta rutina descripta y encripta el texto a mostrar. Puesto que la rutina es la misma tanto para encriptar como para descriptar asumimos que el cifrado es simétrico. Vamos a llamar a esta rutina **"EncryptDecrypt"**.

Analizamos la rutina y vemos que a esta rutina se le pasa el offset a encriptar/descriptar, el size a encriptar/descriptar y un offset que siempre es fijo, **004050A0h**.

Si vamos a ese offset en IDA vemos que son todo 0. Pero con Olly esa zona de memoria está inicializada cuando pasamos por los printf. Vemos que parece tener un tamaño de 256 bytes. Vamos a reiniciar el crackme y en el oep ponemos un BPW (breakpoint hardware de escritura) en **004050A0h**. Rompemos en la rutina de **0040174Ah**. Traceamos/estudiamos la rutina y nos fijamos en el grupo de instrucciones entre **0040175Dh** y **0040177Eh**. Inicializa cada byte del buffer con offset al que corresponde dentro del buffer. El código en C sería:

```
for (counter = 0; counter < 256; counter++)
{
    state[counter] = counter;
}
```

Este código es parte de la inicialización de la key del algoritmo **RC4**. Al contrario que otros algos de cifrado, RC4 no es detectable por signatures y sólo la experiencia de haber trabajado anteriormente con él nos puede ayudar a identificarlo.

En esta rutina también vemos la key para el setup de la RC4Key, **"H15P453C" ;)**

Seguimos. Un poco más abajo, encontramos un call a **00401E1Fh**. En esta rutina se crea un thread cuya entrada es **00401B7Dh**. En esta dirección busca la string **"olldbq"** en RAM ("\\device\\physicalmemory"). En caso de encontrarla llama a una rutina muy simpática: crea un thread recursivo a su propia función, empieza a hacer mallocs hasta dejar el sistema sin memoria. Por último altera la variable global donde se almacena el último CRC realizado por la rutina **CRC32**. Por suerte tenemos un mod modificado de Olly y el crackme no nos detecta ;) (tampoco hay mucho problema en parchear la rutina puesto que el crackme no comprueba que se haya pasado por ella)

Ahora que tenemos los anti solventados, vamos a analizar con tranquilidad la generación de los seriales. El crackme nos pide el **"Name"** y lo transforma en 4 DWORD's que luego los trata como floats de 32bits. Sí, toca mirarse FPU ;)

Después el **"Serial"** lo trata como float y lo divide entre 2. A partir de aquí, realiza una serie de operaciones y para que el serial sea válido se ha de cumplir la siguiente condición:

```
0,00001 > ABS(FloatNameCRC_1 + FloatNameCRC_2 + FloatNameCRC_3 + FloatNameCRC_4)
```

El valor de esas variables, se resume así:

```
FloatNameCRC_4 = FloatNameCRC_4_ORG
FloatNameCRC_3 = FloatNameCRC_3_ORG * (tan Float_Serial)
FloatNameCRC_2 = Var_Float_3 * (tan Float_Serial)
Var_Float_3     = FloatNameCRC_2_ORG * (tan Float_Serial)
Var_Float_1     = FloatNameCRC_1_ORG * (tan Float_Serial)
Var_Float_2     = Var_Float_1 * (tan Float_Serial)
FloatNameCRC_1 = Var_Float_2 * (tan Float_Serial)
```

En mi caso, de "Name" he puesto "**Gadix**" y:

```
FloatNameCRC_1_ORG = 10.0
FloatNameCRC_2_ORG = 1.0
FloatNameCRC_3_ORG = 6.0
FloatNameCRC_4_ORG = -6.0
```

Sustituimos cada FloatNameCRC de la fórmula final por su valor y "tangente Float_Serial" por "X" y nos queda una "bonita" ecuación. Sinceramente no me apetece resolverla o buscarme un programa de mates que me la resuelva. Así que aplico el algoritmo (optimizando el número de accesos a memoria para ganar velocidad, puesto que tarda un ratito en sacar todos los serials válidos) y hago fuerza bruta para sacar un Serial válido. En unos minutos tengo un serial que da "Code OK"

Name: Gadix

Serial: 1.0709353685379028320

He realizado también un keygen que saca todos los serials válidos para el nombre introducido. Con el keygen he tenido unos cuantos dolores de cabeza, puesto que me generaba serials que a priori deberían ser válidos y no lo eran. Todo fue debido porque la mayoría de desensambladores no especifican si la operación FPU tiene precisión de 32 o 64 bits. El desensamblado de la instrucción (la string) es el mismo, pero no el opcode y, en ocasiones, tampoco el resultado. Puesto que el código del keygen está ripeado del crackme, el keygen heredaba el fallo del disassembler. También me he encontrado con problemas con el inline assembler de Visual Studio, y no he encontrado manera de especificar a instrucciones como por ejemplo FADD, si los operandos son **m32real** o **m64real**. La solución ha sido usar "**_emit**" que nos permite hardcodear los bytes de la instrucción. Adjunto el listado de serials válidos para mi nick ("Gadix")