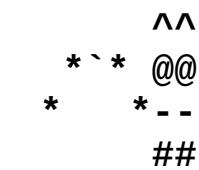
++
Hola listeros quiero compartir esta info con ustedes. Siempre estoy investigando para ir adquiriendo experiencia. Espero les guste.

Iv¡n$0n
++

-[0x08]----------
-[Historia de un CrackMe]----------
-[by blackngel]----------SET-36--



HACK THE WORLD

by blackngel <blackngel1@gmail.com>

<black@set-ezine.org>

(C) Copyleft 2009 everybody

1 - Introducción
2 - Aproximación
3 - Desempacado
4 - Debugging
5 - KeyGen
6 - Conclusión

---[1 - Introducción

En este artículo me propongo el análisis de un CrackMe. Su origen, se me propuso en un "reto". No mencionaré aquí cual es, pues nuestro objetivo es el estudio y no facilitar la fama de aquellos que solo desean obtener un beneficio de reconocimiento.

Esta e-zine ha carecido durante un tiempo de temas relaciones con la ingeniera inversa. Este es un buen momento para volver a la carga. Todos sabemos ya que el cracking es un arte que evoluciona al mismo ritmo en que los fabricantes de software procuran nuevos algoritmos de protección.

Se asumirá un conocimiento básico de las herramientas clásicas como OllyDBG.

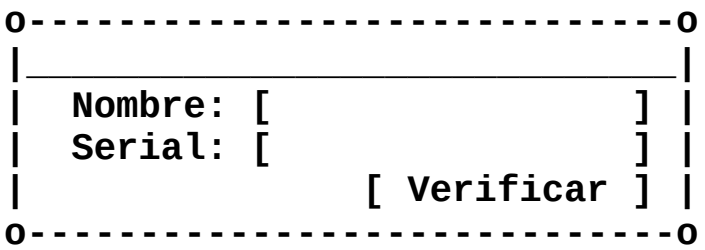
Aunque no es requisito esencial para comprender lo que en este estudio se detalla.

Sin mas, veamos punto por punto como destripar nuestro objetivo.

---[2 - Aproximación

No hay mucho que decir aquí. Lo primero a destacar es que al descomprimir el ejecutable que venia dentro de un "*.rar", el icono que presenta se nos hace familiar. Nada mas y nada menos que el asignado por defecto a los proyectos de Visual Basic. Es un aspecto del que no se puede fiar uno, pero si nos sirve para tener en cuenta.

Si ejecutamos el programa se nos muestra una ventana principal con dos labels y dos cuadro de texto y un boton. Algo asi:



Si introducimos un nombre y serial cualquiera obtendremos un bonito "MsgBox" con el texto "Serial Incorrecto", y cuando pulsemos el botón "Aceptar" se nos echará como a perros del programa.

Que desagradable... pero nosotros tenemos ciertas habilidades para solventar esta situación.

---[3 - Desempacado

Antes de nada, y como todo practicante de ingeniería inversa, necesitaremos un pequeño arsenal de herramientas:

- OllyDBG -> Debugger para programas de Win32
- PeID -> Identificador de Empacadores
- PEditor -> Editor de cabeceras PE
- Java -> Para compilar el KeyGen

Lo primero que a un servidor se le ocurre hacer es abrir el "crackme.exe" con OllyDBG para comprobar si se produce un desensamblado correcto. Compruebo que el programa no comienza con un prologo de funcion clasico sino con algo como esto:

```
pushad
mov esi, Crackme1.0047000
lea edi, dword ptr ds:[esi+ffffa000]
```

Un prologo de funcion normal se veria como alguno de los que aqui se muestran:

```
DELPHI            C               Visual Basic
------------      -----------     ------------
------------
push ebp              push ebp            push
nombre.xxxxxxxx
mov ebp,esp           mov ebp,esp         call
<jmp.&MSVBVM60.#100>
add esp,-xxx      sub esp,x       add byte ptr
ds:[eax],al
```

Bueno, la mayoria de los packers utilizados para cifrar/comprimir un programa utilizan como primer elemento la instruccion de ensamblador "pushad". Esta coloca el contenido de todos los registros en la pila para guardar el estado y poder recuperarlos intactos en cualquier momento.

Si ademas damos un pequeño paseo por la zona "All Referenced Text Strings" apenas veremos algo con sentido. Esto acaba de confirmar lo que nos temiamos.
Lo unico que observamos en la primera cadena es esto:

- "by Hendrix"

Parece que hemos identificado al creador del reto. Pero veamos entonces que ocurre si pasamos el ejecutable a PeID.

Entrypoint: 00007E80 EP Section: rix
File Offset: 00001280 First Bytes: 60,BE,00,70
Linker Info: 6.0 Subsystem: Win32 GUI

Nothing found *

Parece que no hemos tardado en encontrarnos con la primera trampa. Sabemos que el crackme esta empacado de alguna manera pero PeID no quiere reconocer con que algoritmo se ha hecho. Pero tenemos una pista de nuestro lado, y como en toda investigacion no podemos dejarla atras:

EP Section: rix

Que coincidencia! Precisamente las tres ultimas letras del nombre del autor de este CrackMe. Seguramente por diversión ha modificado el nombre de las secciones a mano, lo que no parece afectar a la ejecucion del programa pero si a nuestros intereses.

Si hacemos click sobre la flecha situada al lado de "EP Section", obtendremos más informacion sobre las secciones. Interesante:

```
   Hend     000010000  00006000  00000400  00000000
E0000080
   rix      000070000  00001000  00000400  00001000
E0000040
   .rsrc    000080000  00001000  00001400  00000A00
C0000040
```

Aja! Ahora comprendemos perfectamente que los nombres de las secciones utilizadas por el "packer" han sido modificadas. Solo debemos buscar entre los mas comunes para estudiar cuales son sus nombres reales y asi proceder a su nueva modificacion.

Ya todos conoceis UPX, ¿verdad? Pues bien, echando un vistazo a un programa empacado con este software vemos que sus secciones se denominan asi:

UPX0
UPX1
UPXn (en orden ascendente)

Tambien sabemos que el tamaño de un ejecutable debe permanecer intacto para que este sea ejecutado de manera correcta. Entonces, ¿porque la segunda cadena no posee 4 carácteres? Pues esto no es del todo cierto, si vuelves hacia atrás en PeID y situas el cursor encima de la cadena "rix" y te desplazadas hacia la derecha, comprobaras que existe un espacio final que termina de completar los 4 carácteres necesarios para una correcta modificación.

Utilizaremos ahora la aplicación "UPX Shell (by ION Tek)", que no es mas que una GUI para comprimir o descomprimir ejecutables empacados con este sistema.
Si abrimos el fichero y damos a descomprimir obtenemos lo siguiente:

UPX returned following error: UPX: C:\Crackme1.exe:
CantUnpackException: file is modified/hacked/protected; take care!!

Amigo Hendrix, te hemos cazado. Veamos ahora como trucar nuevamente sus nombres.

Abrimos el crackme con "PEditor" y nos dirigimos al apartado "sections", allí visualizaremos exactamente la misma informacion que en PeID. La diferencia es que podemos hacer click derecho encima de cada seccion para modificar sus nombres por "UPX0" y "UPX1" respectivamente y aplicar los cambios.

Ahora ya podemos volver a UPX Shell para desempacar a este personaje que goza de las travesuras. Podemos ver lo siguiente en la barra de estado:

Crackme1.exe -> File successfully decompressed (in 0,031 seconds)

Claramente pueden ver porque este arte se llama "ingenieria inversa". Estamos deshaciendo uno a uno los pasos que realizo su creador para burlar nuestra inteligencia.

---[4 - Debugging

Excelente, abrimos el CrackMe ya desempacado con OllyDBG y nos encontramos con algo que nos suena de hace un momento. Precisamente un prologo de función clásico de un programa compilado con Visual Basic.

```
00401258 > $ 68 94144000    PUSH Crackme1.00401494
0040125D     . E8 F0FFFFFF          CALL
<JMP.&MSVBVM60.#100>
```

```
00401262     .  0000                  ADD  BYTE  PTR
DS:[EAX],AL
```

En fin... parece que el icono no mentía...

Analicemos las "Referenced Text Strings" para ver que encontramos:

```
 [00402A54] UNICODE "Serial Incorrecto"
 [00402AFC] UNICODE "Serial Correcto"
 [00402D47] UNICODE "Serial Incorrecto"
 [00402DC7] UNICODE "Serial Incorrecto"
```

El primer par se encuentra en direcciones muy cercanas, y lo mismo ocurre con el segundo par. Bien, con esto podemos suponer algunas cosas. Seguramente las dos últimas sean comprobaciones sobre las carácterísticas de nuestro NOMBRE y SERIAL
, de otro modo no tendría sentido mas "Chicos Malos".

Si esto es cierto, significa que las dos primeras son el lugar donde se comprueba realmente si nuestro serial es correcto o no y dependiendo del resultado nos manda un MessageBox u otro.

Empecemos estudiando entonces el último par. Para ello hacemos doble click en la tercera cadena. Si nos desplazamos un poco más hacia arriba desde esa cadena nos encontramos con esta tirada de código:

[-----]

```
00402C98     .  8B1D 10104000   MOV  EBX,DWORD  PTR
DS:[<&MSVBVM60.__vbaLenBstr>]
00402C9E   . 50              PUSH EAX
00402C9F     .  FFD3                    CALL  EBX
<&MSVBVM60.__vbaLenBstr>
.......
.......
.......
00402D1E     .  66:837E  3C  00    CMP  WORD  PTR
DS:[ESI+3C],0
```

```
00402D23   . BB 04000280    MOV EBX,80020004
00402D28      .  75  6C                JNZ  SHORT
Crackme1.00402D96
00402D2A   . BF 0A000000    MOV EDI,0A
.......
.......
.......
00402D47   . C745 9C A41940>MOV DWORD PTR SS:[EBP-
64],Crackme1.00401>;

UNICODE "Serial Incorrecto"
.......
.......
.......
00402D6D      .  FF15  3C104000    CALL  DWORD  PTR
DS:[<&MSVBVM60.#595>] ;

MSVBVM60.rtcMsgBox
.......
.......
.......
00402D94      .  EB  05                JMP  SHORT
Crackme1.00402D9B
00402D96   > BF 0A000000    MOV EDI,0A
00402D9B     >  66:8B56  3C         MOV  DX,WORD  PTR
DS:[ESI+3C]
00402D9F   . 66:6BD2 02     IMUL DX,DX,2
00402DA3   . 0F80 5C010000  JO Crackme1.00402F05
00402DA9      .  66:3956  3E          CMP  WORD  PTR
DS:[ESI+3E],DX
00402DAD      .  74  65                 JE  SHORT
Crackme1.00402E14
.......
.......
.......
00402DC7   . C745 9C A41940>MOV DWORD PTR SS:[EBP-
64],Crackme1.00401>;

UNICODE "Serial Incorrecto"
.......
.......
.......
00402DED      .  FF15  3C104000    CALL  DWORD  PTR
DS:[<&MSVBVM60.#595>] ;
```

MSVBVM60.rtcMsgBox
....
....
....

[-----]

Una vez reducido el código, esto es realmente intuitivo:

Lo primero que vemos es una llamada de VisualBasic para extraer el tamaño de una cadena. Después viene esta comparación:

```
CMP WORD PTR DS:[ESI+3C],0
```

Si hubiéramos colocado un breakpoint en este lugar, veríamos que [ESI+3C] contiene la longitud de nuestro NOMBRE. Entonces sabemos que aquí se comprueba si esta vacio. Si lo esta, nos manda un mensaje de error y nos hecha, en caso contrario salta a esta dirección:

```
MOV EDI,0A                        -> Mete en EDI un 10 decimal
MOV DX,WORD PTR DS:[ESI+3C]       -> Pasa a DX la longitud del nombre
IMUL DX,DX,2                      -> Multiplica por dos la longitud
JO Crackme1.00402F05              -> Comprueba si hay desbordamiento
CMP WORD PTR DS:[ESI+3E],DX       -> Compara longitud SERIAL con longitud NOMBRE * 2
JE SHORT Crackme1.00402E14        -> Si es igual nos deja continuar
```

Como veis, las dos condiciones para que no salte ninguno de los 2 mensajes de error que vimos en "Referenced Text Strings", es que el NOMBRE no este vacio, y que el SERIAL sea el doble de largo que el NOMBRE.

No es que tengamos que intuirlo, pero podemos pensar que por cada carácter de
NOMBRE precisamos 2 en SERIAL.

Una vez conocidas estas condiciones vamos a continuar. Volvemos a las strings y hacemos doble click en el primer "Serial Incorrecto". Muestro nuevamente lo más importante de todo el código que hay a los alrededores:

[-----]

```
ï»¿00402931      . FF15 34104000   CALL DWORD PTR
DS:[<&MSVBVM60.__vbaVarForInit>] ;
00402937    > 3BC3  CMP EAX,EBX
00402939. 0F84 8E010000 JE Crackme1.00402ACD
........
........
........
00402952       .  FF15  A4104000    CALL  DWORD  PTR
DS:[<&MSVBVM60.__vbaI4Var>] ;
00402958     .  8B4B  14           MOV ECX,DWORD PTR
DS:[EBX+14]
........
........
........
00402989       .  FF15  94104000    CALL  DWORD  PTR
DS:[<&MSVBVM60.__vbaStrCopy>]
0040298F     .  8B46  38           MOV EAX,DWORD PTR
DS:[ESI+38]
........
........
........
004029A2       .  FF15  A4104000    CALL  DWORD  PTR
DS:[<&MSVBVM60.__vbaI4Var>]
004029A8     .  8B4B  14           MOV ECX,DWORD PTR
DS:[EBX+14]
........
........
........
004029D9       .  FF15  94104000    CALL  DWORD  PTR
DS:[<&MSVBVM60.__vbaStrCopy>]
004029DF     .  8B45  C8           MOV EAX,DWORD PTR
SS:[EBP-38]
```

```
.........
.........
.........
004029F9      .  FF91  FC060000    CALL  DWORD  PTR
DS:[ECX+6FC]
.........
.........
.........
00402A15   > 66:6BFF 02     IMUL DI,DI,2
00402A19   . 0F80 D6010000  JO Crackme1.00402BF5
00402A1F      .  66:3BBD  38FFFF>CMP  DI,WORD  PTR
SS:[EBP-C8]
00402A26   . 0F84 84000000  JE Crackme1.00402AB0
.........
.........
.........
00402A54   . C785 74FFFFFF >MOV DWORD PTR SS:[EBP-
8C],Crackme1.004019A4
                                          ;
UNICODE "Serial Incorrecto"
.........
.........
.........
00402AB0      > 8D85  14FFFFFF   LEA  EAX,DWORD  PTR
SS:[EBP-EC]
.........
.........
.........
00402AC2       .  FF15  C0104000    CALL  DWORD  PTR
DS:[<&MSVBVM60.__vbaVarForNext>]
00402AC8    .^E9 6AFEFFFF    JMP Crackme1.00402937
00402ACD       >  66:837E  40  FF    CMP  WORD  PTR
DS:[ESI+40],0FFFF
00402AD2       .  75  7A                JNZ  SHORT
Crackme1.00402B4E
.........
.........
.........
00402AFC       .  C785  74FFFFFF  >MOV  DWORD  PTR
SS:[EBP-8C],Crackme1.004019CC
                                              ;
UNICODE "Serial Correcto"
.........
.........
```

```
.........
00402B2A       . FF15 3C104000   CALL DWORD PTR
DS:[<&MSVBVM60.#595>]
                                                 ;
MSVBVM60.rtcMsgBox

[-----]
```

Vayamos por pasos nuevamente. Lo primero que observamos es un bucle tipo "for" que parece recorrer todos los carácteres que componen nuestro NOMBRE. Luego vienen dos llamadas a "StrCopy" que van extrayendo los carácteres de nuestro NOMBRE y SERIAL para hacer posteriores comprobaciones.

Si metemos un nombre y un serial cualesquiera (siempre que el segundo sea el doble de largo que el primero) descubrimos que la magia se encuentra aquí:

```
   CALL DWORD PTR DS:[ECX+6FC] -> Aquí se debe
cocinar el serial
   .........
   IMUL DI,DI,2 -> DI contiene el valor ASCII de
un carácter en NOMBRE, se multiplica por 2

   JO Crackme1.00402BF5         -> Se comprueba si
hay desbordamiento

   CMP DI,WORD PTR SS:[EBP-C8] -> Compara el valor
del carácter de NOMBRE multiplicado por 2 con
[EBP-C8]
   JE Crackme1.00402AB0         -> Si coincide pasa
al siguiente carácter.
```

Tomemos como ejemplo que hemos introducido los siguientes valores:

```
   NOMBRE -> r
   SERIAL -> ab
```

Entonces:

```
   DI       = 72  -> Valor ASCII de "r"
```

DI * 2 = E4 -> En hexadecimal
[EBP-C8] = 97h -> Cocinado en <CALL DWORD PTR DS:[ECX+6FC]>

Y como "E4 != 97", pues nos larga fuera con un desagradable mensaje. Parece entonces que debemos estudiar el CALL que acabamos de mencionar para descubrir de donde sale ese valor "97" almacenado en [EBP-C8].

[-----]

```
00402640    > 55             PUSH EBP
00402641    . 8BEC           MOV EBP,ESP
00402643    . 83EC 0C        SUB ESP,0C
00402646          .   68   26114000          PUSH
<JMP.&MSVBVM60.__vbaExceptHandler>
0040264B      .  64:A1  00000000  MOV  EAX,DWORD  PTR
FS:[0]
........
........
........
004026A7    . BB 01000000    MOV EBX,1
004026AC    . BF 02000000    MOV EDI,2
004026B1     .  8B16              MOV  EDX,DWORD  PTR
DS:[ESI]
........
004026C1       .  FF15  10104000   CALL  DWORD  PTR
DS:[<&MSVBVM60.__vbaLenBstr>]
........
........
........
004026FB       .  FF15  34104000   CALL  DWORD  PTR
DS:[<&MSVBVM60.__vbaVarForInit>]
00402701      .  8B3D  B4104000   MOV  EDI,DWORD  PTR
DS:[<&MSVBVM60.__vbaStrMove>]
00402707      .  8B1D  14104000   MOV  EBX,DWORD  PTR
DS:[<&MSVBVM60.__vbaStrVarMove>]
0040270D    > 85C0           TEST EAX,EAX
0040270F    . 0F84 A7000000  JE Crackme1.004027BC
........
........
........
```

```
00402735       .  FF15  A4104000    CALL  DWORD  PTR
DS:[<&MSVBVM60.__vbaI4Var>]
.........
.........
.........
00402744       .  FF15  4C104000    CALL  DWORD  PTR
DS:[<&MSVBVM60.#632>] ;

MSVBVM60.rtcMidCharVar
0040274A     .  8D4D  A8          LEA  ECX,DWORD  PTR
SS:[EBP-58]
.........
.........
.........
0040276E       .  FF15  24104000    CALL  DWORD  PTR
DS:[<&MSVBVM60.#516>] ;

MSVBVM60.rtcAnsiValueBstr
00402774    .  0FBFD0          MOVSX EDX,AX
00402777       .  8995  44FFFFFF    MOV  DWORD  PTR
SS:[EBP-BC],EDX
0040277D     .  8D8D  64FFFFFF   LEA  ECX,DWORD  PTR
SS:[EBP-9C]
00402783       .  DB85  44FFFFFF    FILD  DWORD  PTR
SS:[EBP-BC]
00402789     .  8D55  DC          LEA  EDX,DWORD  PTR
SS:[EBP-24]
0040278C       .  DD9D  3CFFFFFF    FSTP  QWORD  PTR
SS:[EBP-C4]
00402792       .  DD85  3CFFFFFF    FLD  QWORD  PTR
SS:[EBP-C4]
00402798     .  DC4D  D0           FMUL  QWORD  PTR
SS:[EBP-30]
0040279B     .  DD5D  D0           FSTP  QWORD  PTR
SS:[EBP-30]
0040279E    .  DFE0           FSTSW AX
004027A0    .  A8  0D         TEST AL,0D
004027A2    .  0F85 A1000000  JNZ Crackme1.00402849
.........
.........
.........
004027B1       .  FF15  C0104000    CALL  DWORD  PTR
DS:[<&MSVBVM60.__vbaVarForNext>]
004027B7    .^E9 51FFFFFF    JMP Crackme1.0040270D
```

```
004027BC      > DD45 D0            FLD QWORD PTR
SS:[EBP-30]
004027BF       . FF15 AC104000   CALL DWORD PTR
DS:[<&MSVBVM60.__vbaFpI4>]
004027C5    . 99              CDQ
004027C6    . B9 FF000000     MOV ECX,0FF
004027CB    . F7F9            IDIV ECX
004027CD    . 8BCA            MOV ECX,EDX
........
........
........
```

[-----]

Como llevo diciendo, ejecutar el código paso a paso termina siendo mucho más intuitivo que analizar el código muerto. Seguiremos con los valores de nuestro ejemplo anterior para ver lo que ocurre.

En principio vemos una llamada a "vbaLenBstr". En ese momento en [ESI] se almacena la cadena "ab" (nuestro serial). Pero debe ser consciente que si el serial hubiera sido por ejemplo "abcd", [ESI] seguiría conteniendo "ab". De esto se encarga el bucle exterior que vimos en el código anterior, las llamadas a "StrCopy" van extrayendo 2 carácteres del SERIAL para cada carácter del NOMBRE. Bien, no nos liemos.

Pasada esta llamada comienza un bucle que se encarga de realizar unas operaciones matemáticas con los carácteres del SERIAL (los 2 que forman la pareja). Estas operaciones se hacen a través de la FPU, que es una carácterística de los procesadores y del lenguaje ensamblador que logra una mayor eficiencia en operaciones con números reales.

Veamos el código:

```
CALL    DWORD    PTR    DS:[<&MSVBVM60.#516>]    ;
MSVBVM60.rtcAnsiValueBstr
MOVSX EDX,AX
```

En este instante EDX contiene el valor ASCII de la primera letra de la pareja extraída del serial.

EDX = AX = 61h = 97d -> "a"

.....

```
MOV DWORD PTR SS:[EBP-BC],EDX
LEA ECX,DWORD PTR SS:[EBP-9C]

FILD DWORD PTR SS:[EBP-BC]     -> Se carga "97" en el registro ST0

LEA EDX,DWORD PTR SS:[EBP-24]
FSTP QWORD PTR SS:[EBP-C4]
FLD QWORD PTR SS:[EBP-C4]

FMUL QWORD PTR SS:[EBP-30]     -> Se multiplica por "56" (Constante).

FSTP QWORD PTR SS:[EBP-30]      -> Se guarda el resultado donde antes  se almacenaba la constante anterior  y a la vez en el registro ST7.
FSTSW AX
```

.....

Fácil. Lo que ocurre es que el valor ASCII de la primera letra de "ab" se multiplica por una constante que resulta ser "56".

Al final lo que nos queda es esto:

[EBP-30] = ST7 = 97 * 56 = 5432d = 1538h

Lo que ocurre si seguimos traceando en OllyDBG con la tecla F8, es que el bucle se vuelve a repetir pero esta vez para tratar el segundo carácter. Cuando se ejecutan nuevamente las operaciones en la FPU, hay que tener en cuenta el nuevo valor de [EBP-30]. El resultado de lo ocurrido es el siguiente:

Valor ASCII de "b" = 62h = 98d

98 * [EBP-30] = 98 * 5432 = 532336d = 81F70h

Y hasta aquí se acaba la ejecución de este bucle "for(;;)". Si ahora seguimos traceando llegamos a una última operación. Todo esto lo vemos por los valores que va mostrando OllyDBG:

En "EAX" se almacena nuestro resultado: 532336d = 81F70h

MOV ECX,0FF -> Se mueve a ECX el valor 255d
IDIV ECX -> Se divide EAX entre ECX
MOV ECX,EDX -> Se almacena en ECX el resto de la división

En nuestro caso:

ECX = 255
532336 % 255 = 151d
ECX = 151d = 97h

Aja! Aquí lo tenemos, ¿no recuerdas que estábamos buscando de donde salía el valor "97" hexadecimal que se comparaba con el doble del valor ASCII del carácter contenido en NOMBRE?

Esto quiere decir que aquí acaba el proceso de operaciones. Sigue leyendo ahora la siguiente sección para ver como podemos aprovechar este conocimiento en nuestro beneficio para generar seriales validos para un NOMBRE cualquiera.

---[5 - Keygen

Entonces, ¿cual es la condición para que un serial sea valido con respecto a un nombre?

Respuesta: Para cada carácter de NOMBRE, deben existir dos carácteres en SERIAL cuyo resto de dividir entre 255 el resultado de multiplicar el primer carácter por el segundo y por el valor 56,

sea igual al doble del valor ASCII del carácter de NOMBRE.

Gráficamente seria esto:

W -> Carácter de NOMBRE.
X -> Primer carácter de la pareja de SERIAL.
Y -> Segundo carácter de la pareja de SERIAL.

W * 2 == [(X * 56) * Y] % 255

Como puedes ver, la fórmula no es extremadamente complicada. };-D.
Ya supondrías que obtener el resto de una división se consigue a través del operador modulo.

Una vez obtenida la formula con la que se calcula el SERIAL para nuestro NOMBRE, podemos utilizar la fuerza bruta para ir obteniendo los pares correspondientes a cada carácter del NOMBRE.

He realizado la implementación en lenguaje Java ya que resulta más cómodo y económico a la hora de trabajar con cadenas.

[-----]

```
package keygen_hendrix;

/**
 *
 * @author blackngel
 */
public class Main {

    static String charset =

"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVW
XYZ0123456789";

    public static void main(String[] args) {

       String user = "blackngel";
       int value;
```

```
        int first, second, rest;
        char f, s;
        int found = 0;

        System.out.println("USER: " + user);
        System.out.print("SERIAL: ");

        for(int i = 0; i < user.length(); i++) {

           found = 0;
           value  =  Integer.valueOf(user.charAt(i))
* 2;

           for(int  j  =  0;  j  <  charset.length();
j++) {
              for(int  k  =  0;  k  <  charset.length();
k++) {

                 f = charset.charAt(j);
                 s = charset.charAt(k);

                 first = 56 * Integer.valueOf(f);
                 second   =   Integer.valueOf(s)   *
first;
                 rest = second % 255;

                 if((value  ==  rest)  &&  (found  ==
0)) {
                    found = 1;
                    System.out.print("" + f + s);
                 }
              }
           }
        }
     }
}

[-----]
```

Un punto importante a destacar es el uso de la variable "found". Con esto consigo que una vez encontrado un par coincidente, lo imprima y no lo haga más veces hasta que pase al siguiente carácter del NOMBRE.

El motivo es que existen muchas mas coincidencias, y por lo tanto, muchos seriales validos para un solo nombre. Pero a nosotros con uno nos llega por el momento. Este fue mi resultado:

```
USER:   blackngel
SERIAL: bRcJaRcRazdPbkeRcJ
```

Ten en cuenta entonces que podrías almacenar en una tabla todos los seriales posibles para un nombre e imprimir en pantalla uno aleatorio. Esto daría una sensación más atractiva, aunque nada tiene que ver con nuestro estudio.

Modifica el código anterior para utilizar "args" como entrada y tendrás un KeyGen en toda regla para ese CrackMe en concreto.

---[6 - Conclusión

Como se ha podido comprobar, no hacia falta ser un verdadero experto para llegar a la solución de este crackme. Aunque nunca vienen mal unos conocimientos mínimos de ensamblador, depuración, ingeniería inversa general, astucia y bastante paciencia.

Lo que si muestra muy bien este artículo, es que si tienes la capacidad suficiente como para saber extraer las partes de código que realmente intervienen en el algoritmo de generación/comparación del serial, entonces lo tendrás todo de tu lado para lograr tener éxito.

Lo importante, como siempre, es no tener miedo a enfrentarse con nuevos retos.

Nunca nadie te dirá nada si no los superas, pero la recompensa de haberlo intentado te será pagada con experiencia. Algo que no se adquiere únicamente con un libro en la mano.

Un abrazo, y feliz cracking!

EOF